# 개발자 SQL 튜닝 기본교육

(2009.06.10)

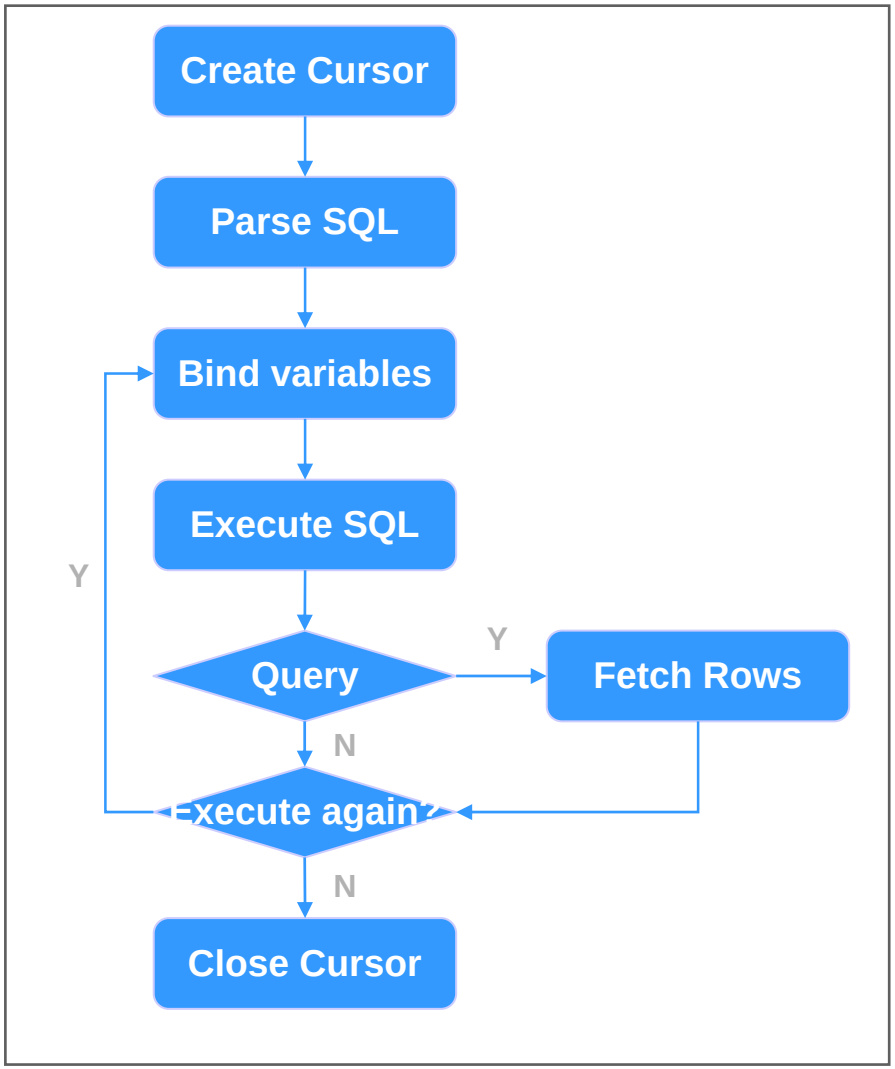
DA

# Revision History

| Version | Updates | Author | Date |
|---|---|---|---|
| 1.0 | Initial Draft | 정윤구 | 2008-03-06 |
| 1.1 | 페이지 처리 SQL 가이드 추가 | 정윤구 | 2009-06-15 |

# 1. SQL 구조 이해

# SQL 처리 구조




### ① Create Cursor
SQL문장 처리를 위한 메모리 일정 영역 점유

### ② Parse SQL
SQL 구문 분석과 최적화를 통한 실행 계획 생성

### ③ Bind Variables
구문 실행 전, Bind 변수 사용한 경우 변수값 대응

### ④ Execute SQL
* DDL, DML 경우 – 구문 자체 실행됨
* 질의(QUERY) – Row Fetch 하기 위한 준비수행

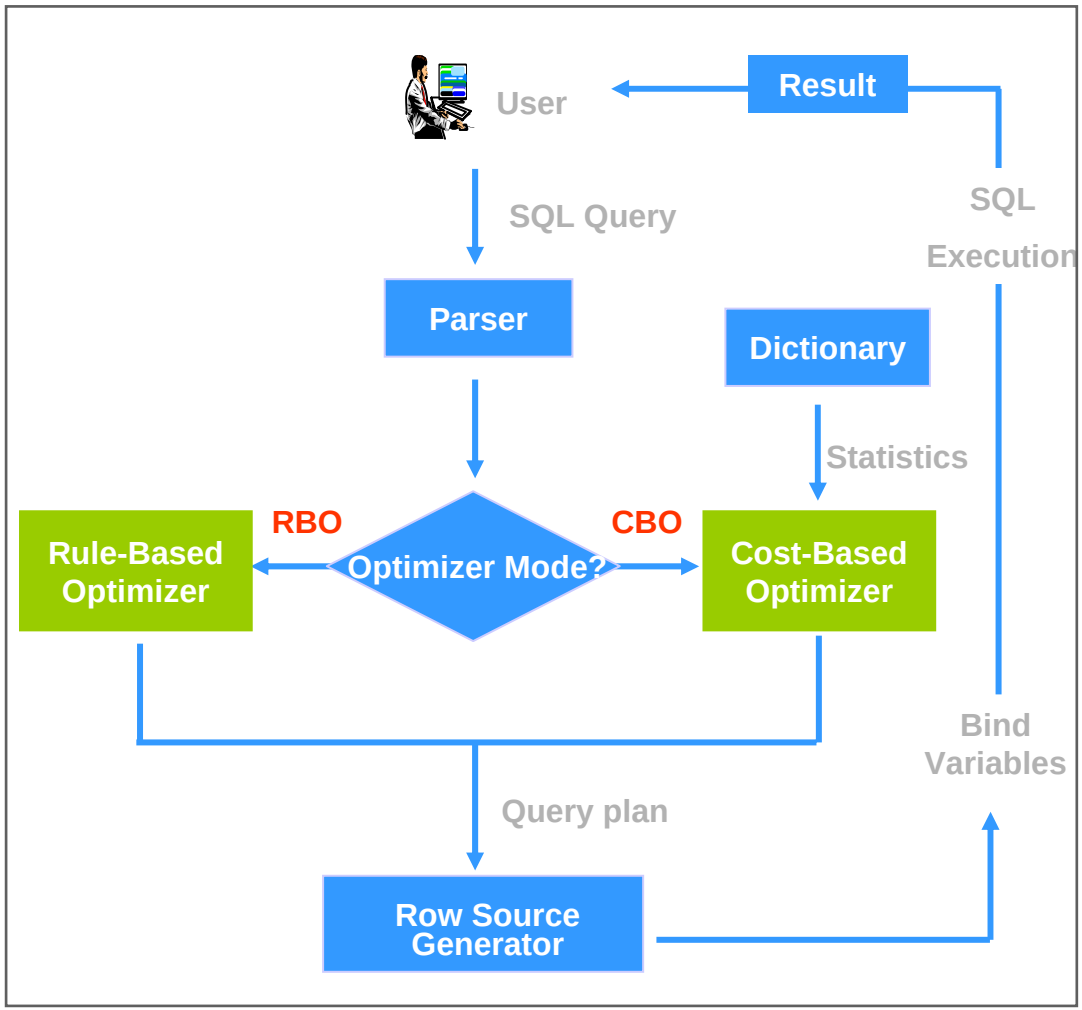
### ⑤ Fetch Rows
Execute 결과값 검색하여 Row 반환

### ⑥ Close Cursor
SQL 처리에 할당된 메모리와 관련된 자원[1]을 반환하며 공유된 자원은 메모리내(Shared Pool 내)에 잔류

*1)공유된 자원 : Parsed SQL, Execution plan*

# OPTIMIZER

Optimizer는 SQL문의 표현과 조건을 평가하여 Data 접근 방법, 조인 방식 등을 결정한다. 이러한 과정을 통해 가장 최적화된 접근 경로(Access Path)를 찾아 실행계획(Execution Plan)을 작성한다.




## ① OPTIMIZER 종류

### (1) RBO

기 정의된 규칙에 따라 실행 계획 생성

### (2) CBO

통계정보 이용하여 최소 비용의 실행 계획 생성

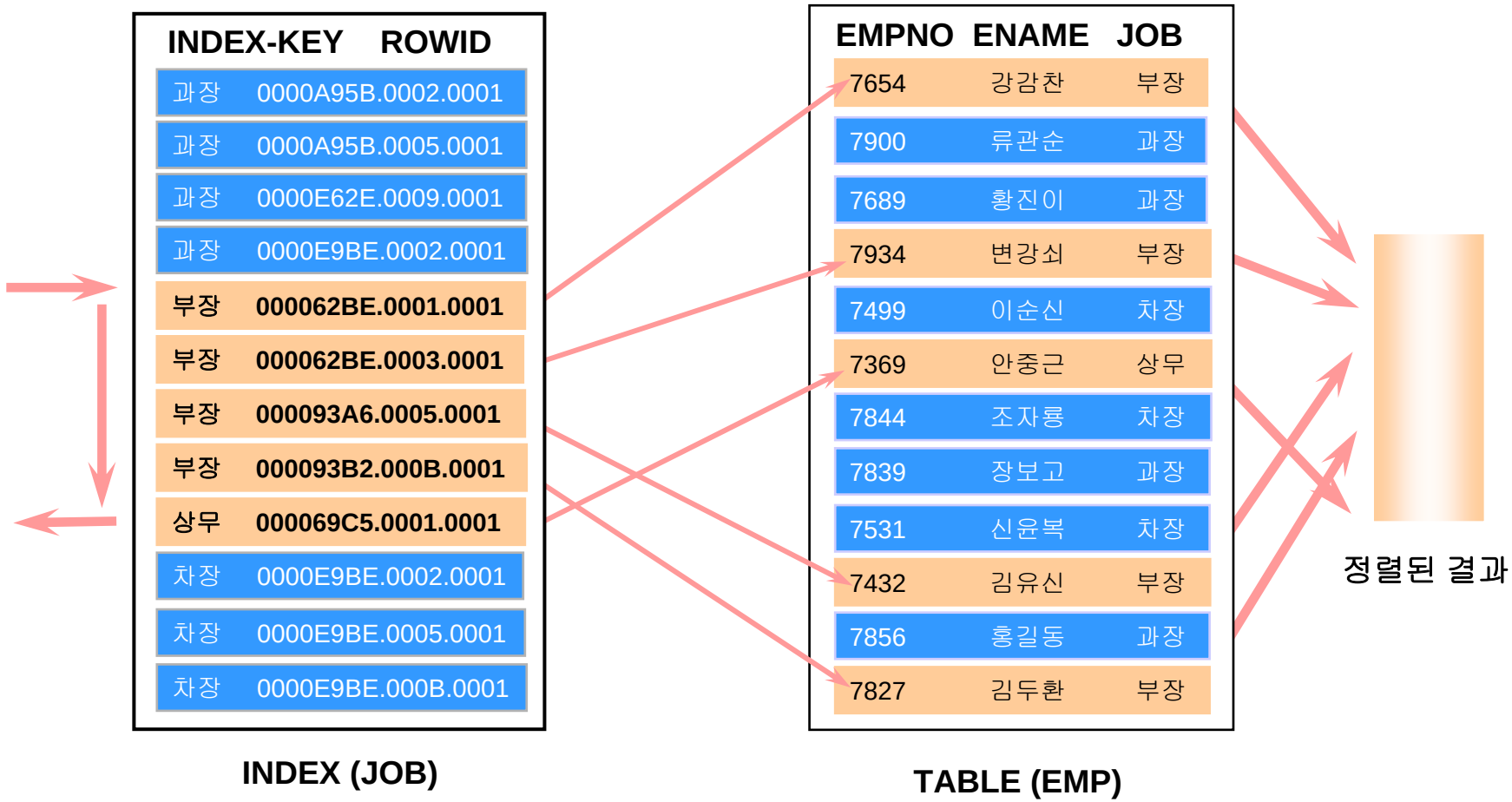
## ② OPTIMIZER MODE

| Optimizer Mode | 설명 |
|---|---|
| **CHOOSE** | 통계정보가 있는 경우 CBO(ALL_ROWS), 통계정보가 없는 경우 RBO 선택 |
| **ALL_ROWS** | Total throughput 기반 최적화 |
| **FIRST_ROWS_n** | First n rows를 가져오는 Fast Response Time 기반 최적화 |
| **FIRST_ROWS** | First row를 가져오는 Fast Response Time 기반 최적화 |
| **RULE** | 기정의된 RULE에 따라 최적화 (Object statistics가 있어도 RBO 선택) |

# 2. 인덱스

## 인덱스의 정의

인덱스는 테이블의 로우와 하나씩 대응되는 별도의 객체를 말하며 인덱스를 생성시킨 키 컬럼과 **ROWID**로 구성되어 정렬된 상태로 저장되어 있음

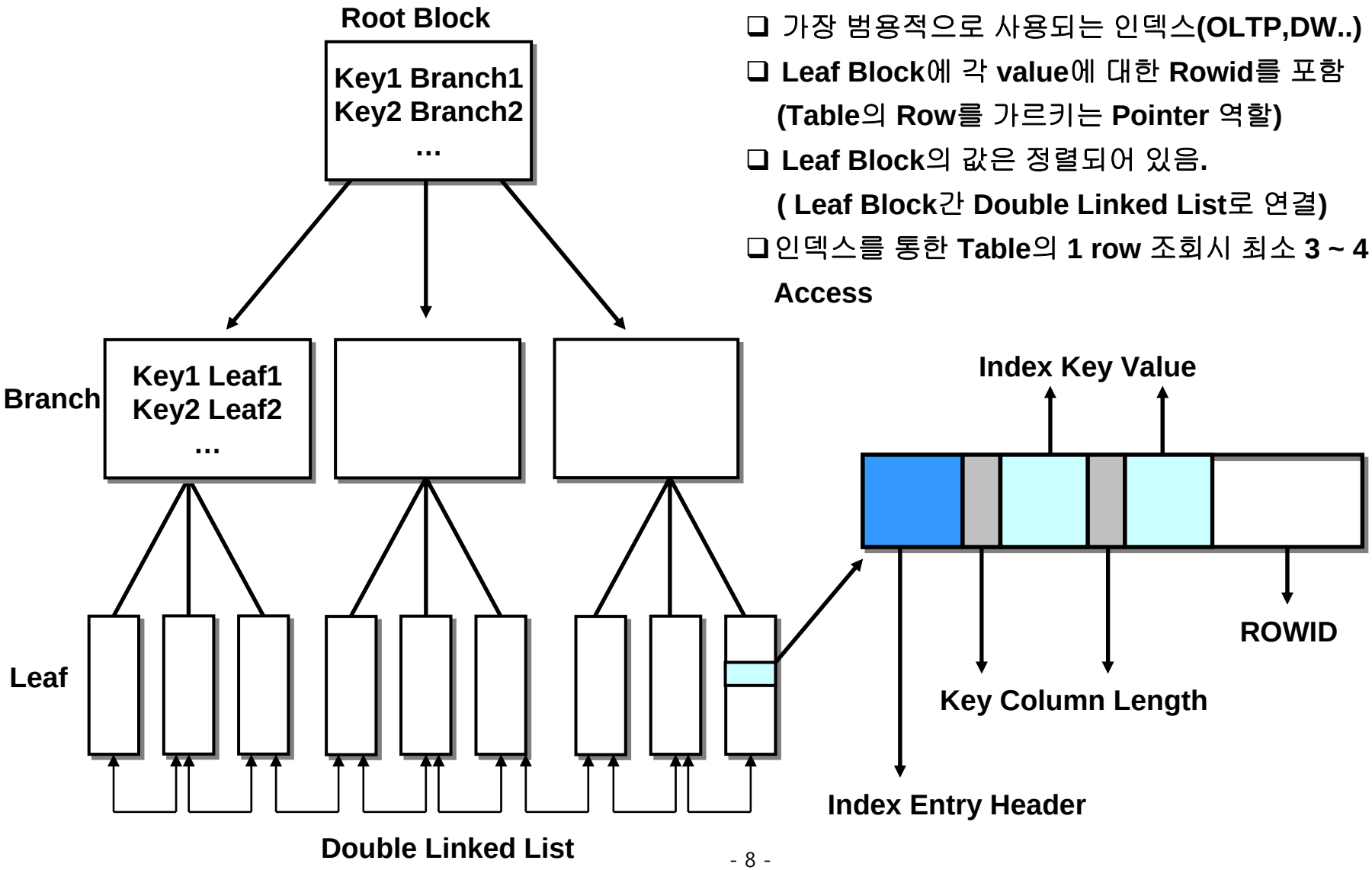


## Index의 종류

**B*Tree, Bitmap, Function-Based, Reverse-Key** 등의 인덱스가 있으며 구문 및 데이터 특성, 분포도 등에 대한 고려 필요

| 인덱스 **Type** | 설명 | 적용 가이드 |
|---|---|---|
| B*Tree | •일반적인 **Normal** 인덱스<br>•**B*Tree** 구조이며 **Leaf Block**에 **Value**에 대한 **Rowid** 저장 | •가장 범용적<br>•인덱스를 구성하는 컬럼의 **Cardinality**가 높은 경우(*동일 값의* ***ROW****가 적은 비율을 가지는 컬럼 즉, 컬럼의 분포도가* ***10 ~ 15 %*** *이내인 경우*) 적합 |
| Bitmap | •**Bitmap** 형태로 **Value**에 대한 **Rowid** 저장<br>•일반적인 인덱스 보다 작은 공간 차지 | •컬럼의 **Cardinality**가 매우 낮은 경우 적합<br>•**DML** 작업시 성능 저하 유발 하므로 주의 필요<br>•**RBO**에서는 불가 |
| Function-Based | •함수(**function**)이나 수식(**expression**)으로 계산된 결과에 대해 인덱스를 생성<br>•인덱스 형태로 존재하는 미리 계산되어 있는 결과를 가지고 처리 | •함수나 수식 사용이 반드시 필요하며 성능 향상을 위해 인덱스가 필요한 경우 적합<br>•**Aggregate Function(SUM,AVG** 등) 에 대한 적용 불가<br>•**RBO**에서는 불가 |

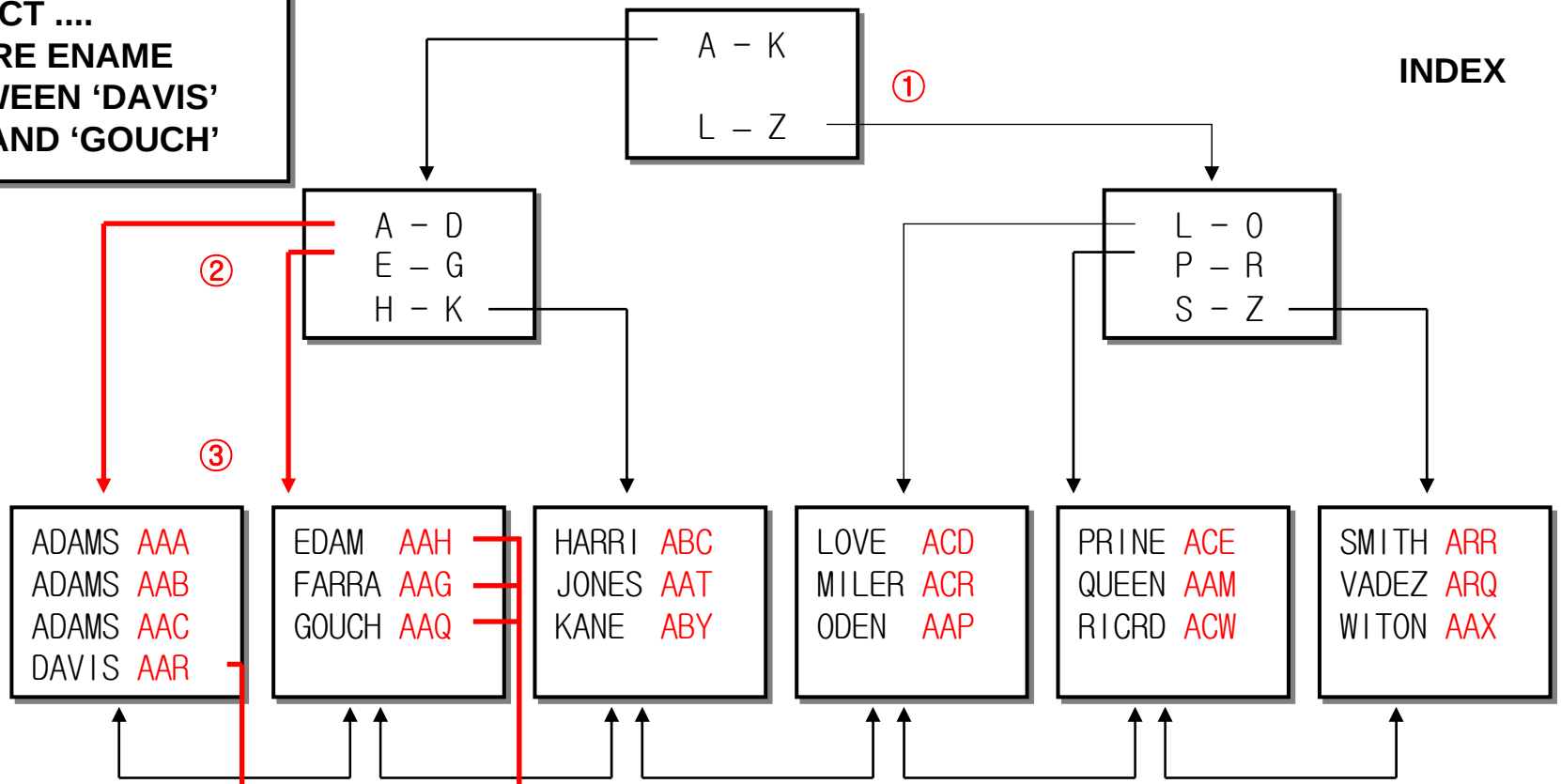
## B*Tree Index




- 가장 범용적으로 사용되는 인덱스(OLTP,DW..)
- Leaf Block에 각 value에 대한 Rowid를 포함 (Table의 Row를 가르키는 Pointer 역할)
- Leaf Block의 값은 정렬되어 있음. ( Leaf Block간 Double Linked List로 연결)
- 인덱스를 통한 Table의 1 row 조회시 최소 3 ~ 4 Block Access

## B*Tree Index

**SELECT ....**
**WHERE ENAME**
**BETWEEN 'DAVIS'**
**AND 'GOUCH'**




| ROWID | AAH | AAG | AAC | AAR | AAA | AAB | AAQ | ABC | AAT | ABY | ACD | ACR | AAP | .... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ENAME | EDAM | FARRA | ADAMS | DAVIS | ADAMS | ADAMS | GOUCH | HARRI | JONES | KANE | LOVE | MILER | ODEN | .... |
| ETC | .... | .... | .... | .... | .... | .... | .... | .... | .... | .... | .... | .... | .... | .... |

## Bitmap Index

```
SELECT  count(*)
FROM    emp
WHERE   gender = 'MAN'
AND     married = 'YES'
```

| empno | gender | married | |
|---|---|---|---|
| 001 | WOMAN | NO | … |
| 002 | WOMAN | NO | … |
| 003 | MAN | YES | … |
| 004 | MAN | NO | … |
| 005 | WOMAN | NO | … |
| 006 | MAN | YES | … |
| … | …. | ….. | … |

Bitmap Function

Bitmap Index

| | | | |
|---|---|---|---|
| gender | = | 'MAN ' | 0 0 1 1 0 1 0 1 1 1 0 0 1 1 1 1 |
| gender | = | 'WOMAN' | 1 1 0 0 1 0 1 0 0 0 0 1 0 0 0 0 |
| married | = | 'YES ' | 0 0 1 0 0 1 0 1 0 1 0 0 0 1 0 1 |
| married | = | 'NO' | 1 1 0 1 1 0 1 0 1 0 0 1 1 0 0 0 |

- ❑ each different key value has its own bitmap
- ❑ each position in the bitmap maps to a possible ROWID
- ❑ if value 1 true, 0 false

## Bitmap Index

### 장점

- ❑ **Cardinality**가 낮은 열에 대해 사용
- ❑ **Small Storage** 사용
- ❑ **AND/OR** 등으로 결합된 복합 조건에 최적
- ❑ 전통적인 **B*_Tree** 인덱스의 단점 해소**(OR, NOT, NULL,..)**
- ❑ 전체 **Null Column**도 **Index**에 저장
- ❑ **Bitmap** 압축을 통한 세그먼트 사이즈 감소**.(NULL**값에 대한**)**
- ❑ **DATAWARE HOUSE**등 대량의 **Data**를 **Read Only Mode**로 사용시에 적당

### 단점

- ❑ **DML** 작업에 취약
- ❑ **Block Level Locking**
- ❑ **Rule Base Optimizer**에서는 사용 못함
- ❑ **Online option(build, rebuild)** 사용 못함

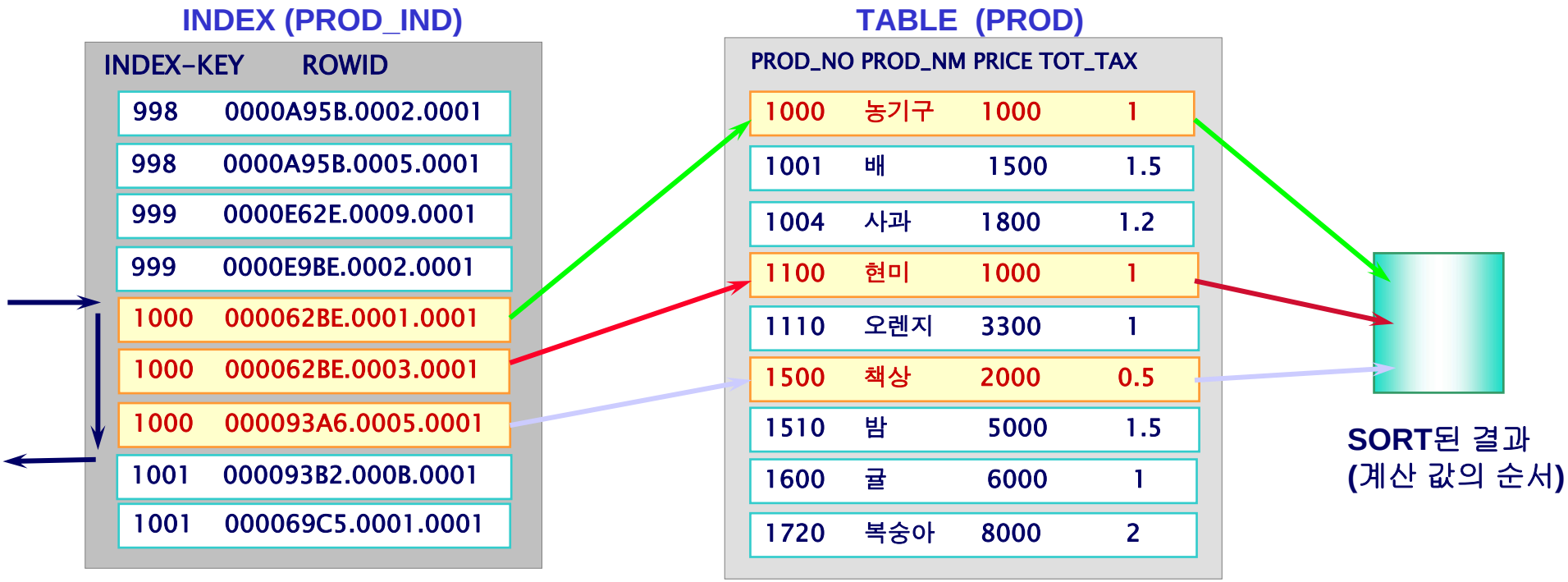
## Function Based Index

**FBI의 구조**

```
SELECT  prod_no,prod_nm,price
  FROM  prod
WHERE  tot_tax * price = 1000
```

```
CREATE INDEX  prod_ind ON
Prod(tot_tax*price);
```

```
TABLE ACCESS BY ROWID PORD
    INDEX RANGE SCAN PROD_IND
```



## Function Based Index

### FBI의 특징

- **Index Column**의 변형에 유연하게 사용할 수 있는 인덱스
  - 함수나 수식의 결과로 **B*Tree** 또는 **Bitmap** 인덱스 생성
  - **CBO** 에서만 사용 가능
  - **Index** 생성 후 통계정보 생성 필수
- **Oralce 8i**부터 사용 가능
- 검색 효율 향상을 위해 효과적이나, **FBI** 구성 컬럼에 대한 빈번한 입력, 수정은 부하 가중
- **SQL** 문에 사용된 **Expression**을 **Parsing**하여 일치하는 **Expression**을 찾고 **Expression Value**를 비교하며, **Expression Value** 에 대해 **Case-Sensitive** 함
- **Dictionary View** 에서 **Index Column** 정보 확인 가능
  - **All(user)_indexes, All(user)_ind_columns, All(user)_ind_expressions**
  - 가공된 **Index Column**은 **System**이 새로운 이름을 부여 **: SYS_NCnnnnn$**

## Function Based Index

### FBI의 제약사항

- PL/SQL 사용자함수는 DETERMINISTIC으로 선언되어야 함
  - Aggregate Function 은 사용 불가
- Object type에 대해서도 FBI 생성이 가능하나 다음의 경우는 불가
  - LOB columns, REF, Nested table column
  - 위의 data type을 포함하는 Objects type
- Parameter Requirement ( + CBO + Analyze )
  - QUERY_REWRITE_INTEGRITY = TRUSTED
    (Oracle 9i 이상에서는 CBO + QUERY_REWRITE_INTEGRITY 만으로 FBI 사
  - QUERY_REWRITE_ENABLED = TRUE
  - COMPATIBLE = 8.1.0.0.0 이상
- User Privilege
  - Index 생성 권한 : INDEX CREATE / ANY INDEX CREATE
  - Query Rewrite 권한 : QUERY REWRITE / GLOBAL QUERY REWRITE
- PL/SQL로 작성된 사용자정의 함수를 사용한 FBI는 Dependency 유지에 주의
  - 인덱스 정의에 사용된 사용자 함수 재정의시 Disabled 로 됨
  - Index Owner의 EXECUTE 권한이 Revoke 되면 Disabled 로 됨
- Optimizer가 Disabled Index 선택시 해당 Query 및 DML 실행은 실패함
  - ALTER INDEX … ENABLE / ALTER INDEX … REBUILD
  - ALTER INDEX … UNUSABLE & SKIP_UNUSABLE_INDEXES = TRUE 로 셋팅
- 함수나 수식의 결과가 NULL 인 경우는 INDEX 사용 불가
- 숫자 컬럼을 문자연산 하거나 문자 컬럼을 수치연산하는 수식의 경우 내부적으로 TO_CHAR, TO_NUMBER가 첨부되어 Expression으로 저장됨에 주의

```
CREATE OR REPLACE FUNCTION
 myUpper(var in VARCHAR2)
RETURN VARCHAR2 DETERMINISTIC
AS
BEGIN
     RETURN UPPER(var);
END;


CREATE INDEX EMP_NAME_IDX
ON EMP(myUpper(ENAME));
```

# 인덱스 생성 기준

## ① 인덱스 생성 컬럼 선정

-SQL문의 WHERE 절에서 상수 조건으로 빈번하게 사용되는 컬럼

-자주 같이 사용되는 조건 컬럼을 결합하여 결합 인덱스 생성

-JOIN의 연결 고리 컬럼

## ② 인덱스 생성시 고려 사항

- 컬럼의 분포도가 10 ~ 15 % 이내인 경우 인덱스 생성( 분포도가 좋지 않은 경우이나 반드시 필요한 경우 Bitmap 인덱스 생성 고려)
- 한 테이블에 인덱스 수가 과다하지 않도록 함(DML 시 오버헤드 발생 및, 유사 인덱스 과다 존재시 성능 저하 발생 우려)
- Unique한 컬럼에 대해서는 Unique 인덱스 생성
- 신규 인덱스 생성시 어플리케이션 상의 영향성 검토 필요
- FOREIGN KEY 컬럼에는 반드시 인덱스를 생성하여 PARENT 테이블에 DML 작업 발생시 CHILD 테이블에 LOCK이 걸리는 것을 방지함 (9i 부터는 불필요)
- 결합인덱스의 선행컬럼은 조건에서 항상 사용 되며 데이터의 분포도가 좋은 컬럼으로 선정
- 선행 컬럼이 '=' 조건으로 비교되지 않는다면 뒤에 있는 컬럼이 '=' 조건으로 사용되었더라도 처리범위는 줄어들지 않으므로 선행컬럼 선정 및 어플리케이션의 조건 대입 주의 필요

# 인덱스 사용 효율성

데이터 모델 과 **SQL** 구문 작성시의 오류 등에 의해 기 존재하는 인덱스의 사용을 방해하거나 인덱스 사용 효율성의 최대화 하지 못하는 경우가 존재함

| 비효율 경우 | 설명 | 권고안 |
|---|---|---|
| 인덱스 컬럼의 외부적 변형 | 인덱스 컬럼에 대한 함수 사용, 연산 수행 등과 같은 외부적 변형 | 인덱스 컬럼에 대한 외부적 변형 제거 |
| 인덱스 컬럼의 내부적 변형 | 데이터 형이 상이한 컬럼에 대한 비교 연산으로 인한 내부적 데이터형 변형 | 데이터 모델상의 정합성 정비 |
| 부정형 비교 | **!=, <> ,NOT IN** 과 같은 부정형 비교 | 문형 변경 등을 통한 부정형 비교 제거 가능 확인 |
| NULL 값 비교 | **IS (NOT) NULL** 비교<br>**(**인덱스에 **NULL** 데이터는 존재하지 않으므로 사용 불가**)** | **= (<>) ‘ ’** 비교로 변경 가능 여부 검토<br>**Default** 값 적용으로 **NULL** 값 제거 가능성 검토 |

# 인덱스 사용 효율성

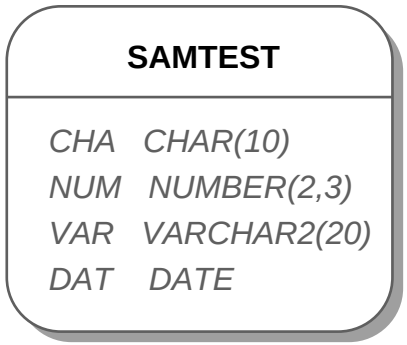
| 비효율 경우 | 설명 | 권고안 |
|---|---|---|
| 결합 인덱스 사용 부적절 | ❑ 결합 인덱스 사용 불가<br>: 선행 컬럼 조건식 누락 혹은 전체범위 비교<br>❑ 결합 인덱스 사용 비효율<br>: 결합 인덱스 구성 컬럼에 대한 조건 누락 및 전체 범위 비교로 인한 결합 인덱스 사용 효율 저하 | ❑ 결합 인덱스 구성 재검토<br>**(컬럼 순서, 컬럼 구성에 대한 검토)**<br>❑ 사용자 입력값에 대한 검토<br>**(필수 입력값에 대한 검토)** |
| LIKE 비교 부적절 | ❑ **DATE** 또는 **NUMBER** 형의 컬럼에 대한 **LIKE** 비교 **(내부적으로 컬럼 변형(character type)** 발생**)**<br>❑ **Like** 조건에 **'%'**로 시작하는 비교값이 들어오는 경우<br>**(LIKE '%text%'** 또는 **LIKE '%text'** 절 **)** | ❑ **BETWEEN** 혹은 **<=** 와 같은 연산자로 변경 고려<br>❑ 사용자 입력값에 대한 검토 |
| 기타 | **Where**절이 아닌 **Having** 절에서 **Filtering** 수행 경우 인덱스 사용 불가 | **Where**절에서 **Filtering** 수행 |

# 인덱스 사용 효율성

① 인덱스 컬럼의 외부적 변형

| 원본 구문 | 외부적 변형 제거 |
|---|---|
| SELECT *<br>FROM EMP<br>WHERE SUBSTR(DNAME,1,3) = 'ABC' | SELECT *<br>FROM EMP<br>WHERE DNAME LIKE 'ABC%' |
| SELECT *<br>FROM EMP<br>WHERE SAL * 12 = 12000000 | SELECT *<br>FROM EMP<br>WHERE SAL = 12000000 / 12 |
| SELECT *<br>FROM EMP<br>WHERE TO_CHAR(HIREDATE, 'YYMMDD') = '940101' | SELECT *<br>FROM EMP<br>WHERE HIREDATE = TO_DATE('940101','YYMMDD') |
| SELECT *<br>FROM EMP<br>WHERE EMPNO BETWEEN 100 AND 200<br>AND NVL(JOB,'X') = 'CLERK' | SELECT *<br>FROM EMP<br>WHERE EMPNO BETWEEN 100 AND 200<br>AND JOB = 'CLERK' |
| SELECT *<br>FROM EMP<br>WHERE DEPTNO \|\| JOB = '10SALESMAN' | SELECT *<br>FROM EMP<br>WHERE DEPTNO = '10' AND JOB = 'SALSMAN' |

② 인덱스 컬럼의 내부적 변형




| 원본 구문 | 내부적 변형 발생 사항 | 발생 이유 |
|---|---|---|
| **SELECT * FROM SAMPLET**<br>**WHERE CHA = 10** | **SELECT * FROM SAMPLET**<br>**WHERE TO_NUMBER(CHA) = 10** | **'CHA'**의 컬럼형은 **CHAR** 이나 조건 비교를 **NUMBER(=10)**로 수행하여 내부적으로 **TO_NUMBER** 함수 적용됨 |
| **SELECT * FROM SAMPLET**<br>**WHERE VAR = 10** | **SELECT * FROM SAMPLET**<br>**WHERE TO_NUMBER(VAR) = 10** | **'VAR'**의 컬럼형은 **VARCHAR2** 이나 조건 비교를 **NUMBER(=10)**로 수행하여 내부적으로 **TO_NUMBER** 함수 적용됨 |
| **SELECT * FROM SAMPLET**<br>**WHERE NUM LIKE '9410%'** | **SELECT * FROM SAMPLET**<br>**WHERE TO_CHAR(NUM) LIKE '9410%'** | **'NUM'**의 컬럼형은 **NUMBER**이나 조건 비교를 **STRING(LIKE '9410%')**로 수행하여 내부적으로 **TO_CHAR** 함수 적용됨 |

③ 부정형 비교

```
SELECT  'Not found !'  INTO  :COL1
  FROM   EMP
 WHERE  EMPNO <> '1234'
```

```
SELECT  'Not found'  INTO :COL1
  FROM   DUAL
 WHERE  NOT EXISTS ( SELECT ''  FROM EMP
                     WHERE EMPNO = '1234')
```

효율적 (?)

```
SELECT  *
  FROM  EMP
WHERE ENAME LIKE '천%'
    AND JOB <> 'SALES'
```

```
SELECT  *
   FROM  EMP a
 WHERE a.ENAME LIKE '천%'
     AND NOT EXISTS ( SELECT ''  FROM  EMP b
                      WHERE a.Empno = b.Empno
                        AND  b.JOB = 'SALES')
```

효율적 (?)

```
SELECT  *
  FROM   EMP
 WHERE  ENAME LIKE '천%'
MINUS
 SELECT  *
   FROM  EMP
 WHERE JOB = 'SALES'
```

④ NULL 값 비교

| 원본 구문 | 개선 사항 | 비고 |
|---|---|---|
| **SELECT * FROM EMP WHERE ENAME IS NOT NULL** | **SELECT * FROM EMP WHERE ENAME > ' ' /* SPACE */** | **ENAME**은 문자열이므로 **NOT NULL** 비교를 ' ' 로 교체 가능 |
| **SELECT * FROM EMP WHERE COMM IS NOT NULL** | **SELECT * FROM EMP WHERE COMM > 0** | **COMM** 은 숫자형이고 양수값만 존재한다는 가정하에 **>0** 으로 교체 가능 |
| **SELECT * FROM EMP WHERE COMM IS NULL** | **CREATE TABLE EMP (…. COMM NUMBER DEFAULT 0, …) /* COMM = 0 */** | **EMP** 테이블의 **COMM** 컬럼내의 **NULL** 값 제거를 수행하기 위하여 테이블 생성시 **Default** 값 지정 **(NULL** 비교 불필요**)** |

## ⑤ 결합 인덱스 순서

```
SELECT *
FROM TAB1
WHERE COL1 = 'A'
 AND COL2 = '113'
```

**CASE1. COL1 + COL2**

| COL1 | COL2 |
|---|---|
| A | 110 |
| A | 111 |
| A | 112 |
| A | 113 |
| A | 114 |
| A | 115 |
| A | 116 |
| A | 117 |
| A | 118 |
| A | 119 |
| A | 120 |
| A | 121 |
| B | 110 |
| B | 111 |

**CASE2. COL2 + COL1**

| COL2 | COL1 |
|---|---|
| 110 | A |
| 110 | B |
| 110 | C |
| 110 | D |
| 111 | A |
| 111 | B |
| 111 | C |
| 111 | D |
| 112 | A |
| 112 | B |
| 112 | C |
| 112 | D |
| 113 | A |
| 113 | B |

## ⑤ 결합 인덱스 순서

```
SELECT * FROM TAB1
WHERE COL1 = 'A'
AND   COL2 between '111'
                and '113'
```

**CASE1. COL1 + COL2**

| COL1 | COL2 |
|---|---|
| A | 110 |
| A | 111 |
| A | 112 |
| A | 113 |
| A | 114 |
| A | 115 |
| A | 116 |
| A | 117 |
| A | 118 |
| A | 119 |
| A | 120 |
| A | 121 |
| B | 110 |
| B | 111 |

**CASE2. COL2 + COL1**

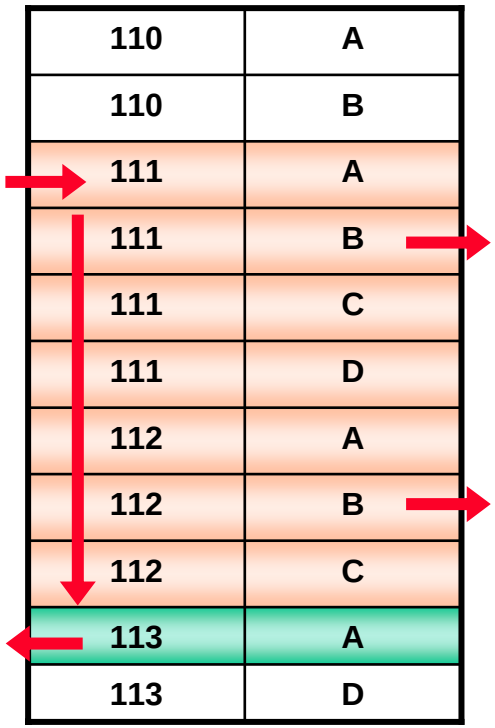
| COL2 | COL1 |
|---|---|
| 110 | C |
| 110 | D |
| 111 | A |
| 111 | B |
| 111 | C |
| 111 | D |
| 112 | A |
| 112 | B |
| 112 | C |
| 112 | D |
| 113 | A |
| 113 | B |
| 113 | C |
| 114 | A |

## ⑥ 결합 인덱스 Access 방식

### CASE1. BETWEEN 사용

**SELECT * FROM item**

**WHERE col1 = 'B' and col2 between '111' and '112'**

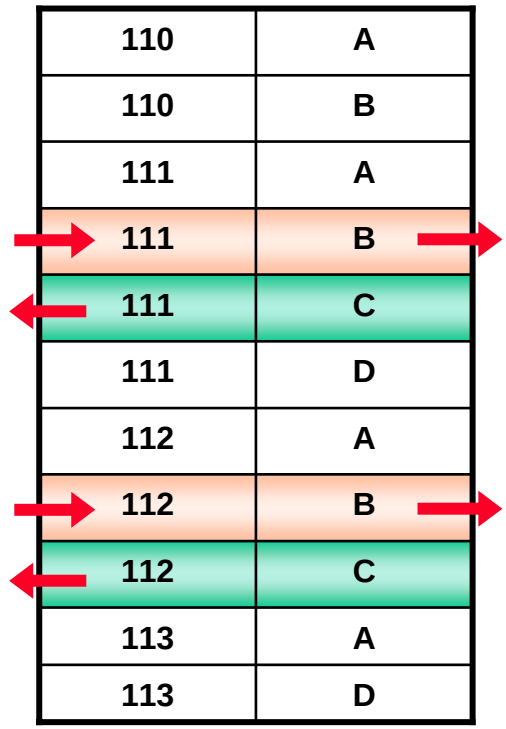


| | |
|---|---|
| 110 | A |
| 110 | B |
| 111 | A |
| 111 | B |
| 111 | C |
| 111 | D |
| 112 | A |
| 112 | B |
| 112 | C |
| 113 | A |
| 113 | D |

### CASE2. IN 사용

**SELECT * FROM item**

**WHERE col1 = 'B' and col2 in ('111' , '112')**



| | |
|---|---|
| 110 | A |
| 110 | B |
| 111 | A |
| 111 | B |
| 111 | C |
| 111 | D |
| 112 | A |
| 112 | B |
| 112 | C |
| 113 | A |
| 113 | D |

# 3. 실행계획

## 실행계획 확인

PLAN TABLE 및 TRACE 파일을 이용한 다양한 방법의 실행계획 확인 가능.

실행계획 확인 방식

| 방 식 | 설명 |
|---|---|
| EXPLAIN PLAN FOR SQL | → SQL Prompt 및 10g 이전버젼의 클라이언트 환경에서 사용.<br>→ 예측 실행계획 확인 가능. |
| SET AUTOTRACE | → SQL Prompt 에서 확인.<br>→ 실행계획 및 간단한 수행통계 확인 가능. |
| SQL TRACE | → 세션 혹은 시스템 전체에 수행.<br>→ 실행계획(예측) 및 수행계획(실제) 확인 가능.<br>→ 세그먼트별 수행정보 확인 가능. |
| EVENT TRACE - 10046 | → 일반적인 SQL TRACE 와 동일.<br>→ SQL Prompt, DBMS_SYSTEM.SET_EV( ) 로 수행 가능.<br>→ 바인드변수, 수행정보, WAIT정보 확인 가능. |
| EVENT TRACE - 10053 | → Oracle Optimizer 수행 정보 확인 가능.<br>→ 10g 이후 SQL Tranformation, Optimizing Low Level 상세 정보 제공. |
| DBMS_XPLAN | → 10g 이후 'EXPLAIN PLAN FOR ' SQL 기본 처리를 대체.<br>→ 10g 이후 다양한 시스템 및 SQL 통계 수집과 통합되어 강력한 기능 제공.<br>→ SQL 실행계획 및 Access/Filter 조건 상세 확인 가능.<br>→ 'SQL TRACE' 와 거의 동일한 수행 정보 제공. |

➔ **SQL Optimizing** 상세 확인 시 **10053 Event Trace** 가 최적의 정보 제공.
➔ 개별 **SQL Tuning** 시 **DBMS_XPLAN** 가 유리.

## EXPLAIN PLAN 이용

PLAN 테이블(PLAN_TABLE)을 이용하여 SQL*Plan 상에서 실행계획 확인

**1. Plan Table 생성**

SQL>@?/rdbms/admin/utlxplan.sql

**2. 실행 계획 생성**

```
EXPLAIN PLAN [SET STATEMENT_ID
= 'text'] [INTO [schema .] table [@ dblink]]
FOR statement;
```

**3. 실행 계획 확인**

SQL>@?/rdbms/admin/utlxpls.sql

```
SQL*Plus > explain plan set statement_id = 'a1'  for
                select  col3, sum(col4) from tab1
                where a.col1 in ('10', '50') group by col3
SQL*Plus > @?/rdbms/admin/utlxpls.sql

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
| Id  | Operation                    |  Name |Rows|Bytes| Cost|
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |       |    |     |     |
|*  1 |  TABLE ACCESS BY INDEX ROWID| EMP   |    |     |     |
|   2 |   NESTED LOOPS               |       |    |     |     |
|   3 |    TABLE ACCESS FULL         | DEPT  |    |     |     |
|*  4 |    INDEX RANGE SCAN          | IDX_02|    |     |     |
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("E"."SAL">1000)
   4 - access("E"."DEPTNO"="D"."DEPTNO")

Note: rule based optimization
```

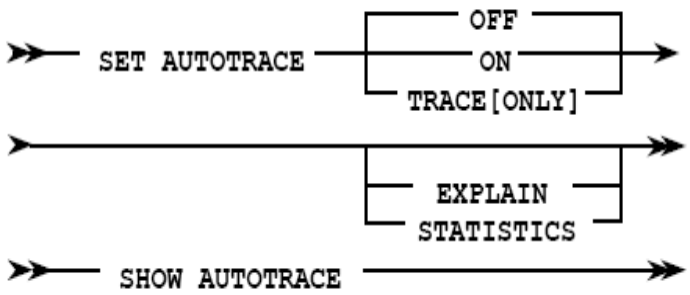
## AUTOTRACE 이용

PLAN 테이블(PLAN_TABLE)을 이용하여 SQL*Plan 상에서 구문 실행 계획 및 통계정보 등 간략한 리포트 자동 생성

### 1. Plan Table 생성

```
SQL>@?/rdbms/admin/utlxplan.sql
```

### 2. AUTOTRACE 상태 설정




```
SQL*Plus > SET AUTOTRACE ON STAT
SQL*Plus > select e.ename, d.dname  from emp e, dept d
           where e.sal > 1000  and e.deptno = d.deptno;

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
   1    0   TABLE ACCESS (BY INDEX ROWID) OF 'EMP'
   2    1     NESTED LOOPS
   3    2       TABLE ACCESS (FULL) OF 'DEPT'
   4    2       INDEX (RANGE SCAN) OF 'IDX_02' (NON-UNIQUE)
Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
         14  consistent gets
          0  physical reads
          0  redo size
        773  bytes sent via SQL*Net to client
        655  bytes received via SQL*Net from client
          2  SQL*Net roundtrips to/from client
          0  sorts (memory)
          0  sorts (disk)
         12  rows processed
```

## Query Tool 사용

**Orange** 사용하여 **Plan** 보기

**Plan보기**
단축키 **:F5**


Orange - [Plan

File Edit View Tools DBA Window Help

SQL

Parsing Schema: <DEFAULT> Refresh Optimizer Mode: <DEFAULT>

Original *

```
1 SELECT *
2 FROM   DEPT A, EMP B
3 WHERE  A.DEPTNO = B.DEPTNO
4 AND    A.DNAME = :V_DNAME
5 AND    B.ENAME = :V_ENAME
6 ;
```

**Access Predicates**
드라이빙 조건 표시

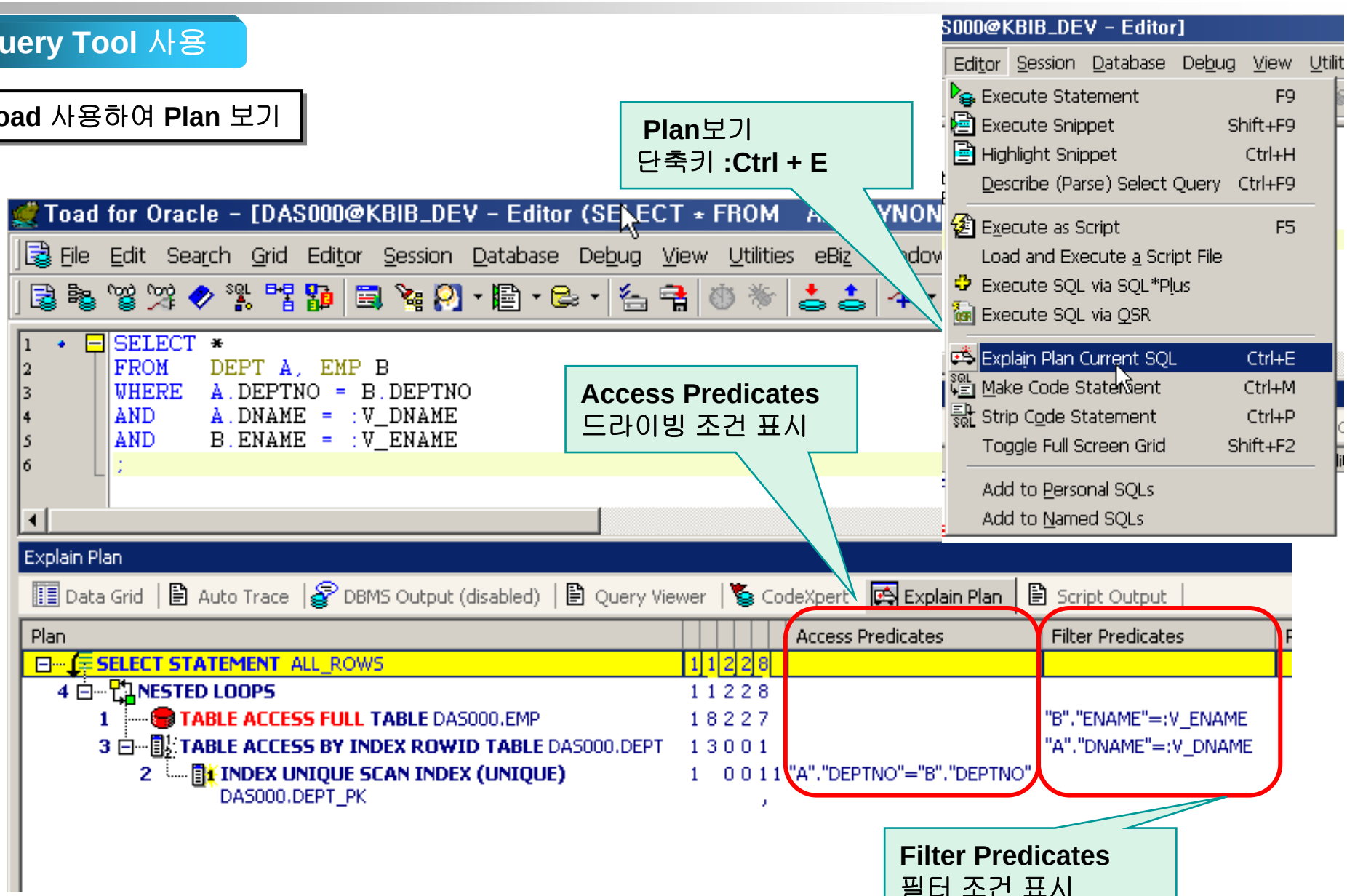
**Filter Predicates**
필터 조건 표시

Plan | Plan | Plan | Schema | Trace | Result

| ID | Operation | Access Predicates | Filter Predicates | Other |
|---|---|---|---|---|
| 0 | SELECT STATEMENT Optimizer=ALL_ROWS (Cost=2 Card=1 Bytes=117) | | | |
| 1 0 | NESTED LOOPS (Cost=2 Card=1 Bytes=117) | | | |
| 2 1 | TABLE ACCESS (FULL) OF 'EMP' (TABLE) (Cost=2 Card=1 Bytes=87) | | "B"."ENAME"=:V_ENAME | |
| 3 1 | TABLE ACCESS (BY INDEX ROWID) OF 'DEPT' (TABLE) (Cost=0 Card=1 B | | "A"."DNAME"=:V_DNAME | |
| 4 3 | INDEX (UNIQUE SCAN) OF 'DEPT_PK' (INDEX (UNIQUE)) (Cost=0 Card=1 | "A"."DEPTNO"="B"."DEPTNO" | | |

## Query Tool 사용

**Toad** 사용하여 **Plan** 보기



## Trace 파일 이용

**SQL*Trace** 를 통해 만들어진 트레이스 파일을 **tkprof** 유틸리티를 이용하여 **Format**

**1. SQL*Trace 활성화 (현재 접속된 세션 상에서 trace 활성화)**

- **ALTER SESSION SET SQL_TRACE = TRUE;**
- **EXEC DBMS_SESSION.SET_SQL_TRACE( TRUE);**
- **ALTER SESSION SET EVENTS '10046 TRACE NAME CONTEXT FOREVER, LEVEL #'**
  - *Trace Level 1 : 기본정보*
  - *Trace Level 4 : 기본정보 + Binding 정보 출력*
  - *Trace Level 8 : 기본정보 + Waiting 정보 출력*
  - *Trace Level 12 : 기본정보 + Binding + Waiting 정보 출력*

**2. Tkprof 실행**

**tkprof tracefile outputfile [options]**

| Option | 설명 |
|---|---|
| **SORT = option** | 명령문 정렬 순서**(ex. execpu :**실행에 사용된 **cpu**시간으로 정렬**)** |
| **EXPLAIN =username/password** | 지정된 **schema**에서 **EXPLAIN PLAN**을 실행함 |
| **SYS = NO** | **SYS user**에 의해 실행된 **Recursive SQL**문의 나열 비활성화 |
| **AGGREGATE = NO** | 다른 **user**의 동일한 **SQL**문을 하나의 레코드로 집계하지 않음 |
| **WAITS = YES** | **Trace**파일에서 발견된 모든 **Wait**이벤트에 대한 요약 기록 여부 |

## Trace 파일 이용

**SQL*Trace** 를 통해 만들어진 트레이스 파일을 **tkprof** 유틸리티를 이용하여 **Format**

### 3. TKPROF 결과 해석

| 단계 | 설명 |
|---|---|
| **PARSE** | **SQL** 구문 분석에서 발생하는 통계치 |
| **EXECUTE** | 명령문을 실행하면서 발생하는 통계치 |
| **FETCH** | **fetch**시에 발생하는 통계치**(select**문이 실행되면서 발생하는 통계치**)** |

| 단계 | 설명 |
|---|---|
| **COUNT** | 각 처리 단계별 실행된 횟수 |
| **CPU** | 각 처리 단계별 **CPU** 소모 시간**(**초**)** |
| **Elapsed** | 각 처리 단계의 시작에서 종료까지 총 경과 시간**(**초**)** |
| **Disk** | 각 처리 단계별 물리적인 디스크 블록 접근 횟수 |
| **Query** | 각 처리 단계별 읽은 변경된 버퍼 블록 수 |
| **Current** | 각 처리 단계별 현 세션에만 유효한 버퍼 블록을 접근한 수 |
| **Rows** | 각 처리 단계별 읽은 총 행수 |

## Trace 파일 이용

### SQL Trace 파일 Sample

**SQL*Plus > alter session set sql_trace=true;**
SQL*Plus > @test.sql

**# tkprof ora_09136.trc output.prf sys=no explain=scott/tiger**

```
 SELECT e.ename, d.dname
 FROM   emp e, dept d
 WHERE  e.sal > 1000 AND    e.deptno = d.deptno

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.02          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.00       0.00          0          3          8          12
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.01       0.02          0          3          8          12

Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 20  (SCOTT)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
      0   HASH JOIN
      0    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'DEPT'
      0    TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'EMP'
```

## Trace 파일 이용

### Oracle 9.2 Event 10046 Segment-level Statistics

- **Oracle 9iR2 New Feature로 추가.**
- **실행계획의 각 단계별, 개별 처리별 수행 정보(Stat) 표현.**
  - 처리 결과만 표현되는 Trace 정보에 대한 보완.
  - 개별 엑세스는 각 단계의 처리 정보를 표시.
  - JOIN 엑세스는 해당 단계까지의 누적 처리 시간을 표시.
- **예) : ALTER SESSION SET STATISTICS_LEVEL = ALL ;**

```
SELECT G.CUST_NO, G.NAME, G.JUMIN_NO, G.SILMYUNG_YMD, M.JUKSU_YMD, M.NOWAMT
  FROM CM_GIBON G, JH_MASTER M
 WHERE G.NAME IN ('홍길동')
   AND G.CUST_NO = M.CUST_NO

Rows     Row Source Operation
-------  ---------------------------------------------------
      2  TABLE ACCESS BY INDEX ROWID JH_MASTER (cr=7 r=0 w=0 time=354 us)
      4   NESTED LOOPS  (cr=5 r=0 w=0 time=236 us)
      1    TABLE ACCESS BY INDEX ROWID CM_GIBON (cr=3 r=0 w=0 time=133 us)
      1     INDEX RANGE SCAN CM_GIBON_NAME (cr=2 r=0 w=0 time=89 us)
      2    INDEX RANGE SCAN JH_MASTER_CUST_NO (cr=2 r=0 w=0 time=49 us)
```

- cr= : logical I/O for consistent reads.
- r= : physical reads.
- w= : physical writes.
- tims= : elapsed time and the timing precision (e.g. us Microseconds)

## DBMS_XPLAN

SQL 실행계획 확인 및 수행된 SQL 수행정보 확인 시 사용.

**1. DBMS_XPLAN 수행**

- **DBMS_XPLAN.DISPLAY : PLAN TABLE 정보 Display**
- **DBMS_XPLAN.DISPLAY_AWR : AWR 저장 SQL/Execution Plan 정보 Display**
- **DBMS_XPLAN.DISPLAY_CURSOR : SGA Loaded SQL Cursor 정보 Display**
- **DBMS_XPLAN.DISPLAY_SQL_PLAN_BASELINE : BASELINE SQL 기준(11g~)**
- **DBMS_XPLAN.DISPLAY_SQLSET : SQL TuningSet 저장 SQL/Execution Plan 정보 Display**

일반적인 사용법 :

- **SELECT * FROM ( DBMS_XPLAN.DISPLAY ) ;**
- **SELECT * FROM ( DBMS_XPLAN.DISPLAY_CURSOR ) ;**
- **SELECT * FROM ( DBMS_XPLAN.DISPLAY_CURSOR(NULL,NULL,'TYPICAL' ) ;**
- **SELECT * FROM**
  **( DBMS_XPLAN.DISPLAY(PLAN_TABLE#, SQL_ID#,FORMAT_OPTION#, FILTER_PREDS#) ;**
- **SELECT * FROM**
  **( DBMS_XPLAN.DISPLAY_CURSOR(SQL_ID#,CHILD_NUMBER#,FORMAT_OPTION#) ;**

## DBMS_XPLAN

FORMAT OPTION 을 사용 상세 정보 확인 가능.

**1. FORMAT OPTION#  (DISPLAY_CURSOR 기준)**

- **BASIC** : 최소 실행계획 정보 제공.
- **TYPICAL** : 기본값(Default), 실행계획 및 **ROWS, BYTES, COST, Temp Space, Predicate** 제공.
- **SERIAL** : **TYPICAL** 과 유사, **PARALLEL** 정보 미 제공.
- **ALL** : **TYPICAL** 제공 정보 + **PROJECTION, ALIAS, REMOTE SQL** 등 제공.
- **ADVANCED** : **ALL** 제공 정보 + **Peeked Binds, Outline, Note** 등 제공.

- **FUNCTION PARAMETER**
  - *ROWS, BYTES, COST, PARTITION, PARALLE* : *기본 실행계획*
  - *PREDICATE, PROJECTION, REMOTE* : *SQL 엑세스 패스 확인, 쿼리 블록, DBLINK 전달 SQL 등 확인.*
  - *IOSTATS* : *I/O 관련 Read/Write 블록 수*
  - *MEMSTATS* : *Hash/Sort 작업등에 사용한 메모리 정보*
  - *ALLSTATS* : *IOSTATS + MEMSTATS*
  - *LAST* : *Default, 최종 수행 정보 기준.*
  - *ALIAS, NOTE, PEEKED_BINDS, …*

## DBMS_XPLAN

일반 업무별 사용 방법.

**1.** 사용 **(DISPLAY_CURSOR** 기준**)**

- 실행 통계 수집 **/*+ GATHER_PLAN_STATISTICS */**
- 단순 실행계획 확인

```
SELECT * FROM TABLE ( DBMS_XPLAN.DISPLAY )
SELECT * FROM TABLE ( DBMS_XPLAN.DISPLAY_CURSOR(NULL,NULL,'TYPICAL') ;
```

- 특정 **SQL** 실행계획 확인

```
SELECT * FROM TABLE ( DBMS_XPLAN.DISPLAY_CURSOR(S.SQL_ID, S.CHILD_NUMBER,'TYPICAL') ) ;
```

- **SQL** 실행계획 상세 확인

```
SELECT * FROM TABLE ( DBMS_XPLAN.DISPLAY_CURSOR(NULL,NULL,'ALLSTATS LAST') ) ;
```

- **SQL** 실행정보 상세 확인 ← **SQL TRACE** 대체 사용 가능**!!!**

```
SELECT * FROM TABLE ( DBMS_XPLAN.DISPLAY_CURSOR(NULL,NULL,'ADVANCED ALLSTATS LAST') ) ;
```

- **BIND** 변수 확인

```
SELECT * FROM TABLE ( DBMS_XPLAN.DISPLAY_CURSOR(NULL,NULL,'ALLSTATS LAST  +PEEKED_BINDS') ) ;
```

- **OUTLINE** 정보 확인

```
SELECT * FROM TABLE ( DBMS_XPLAN.DISPLAY_CURSOR(NULL,NULL,'OUTLINE') ;
```

## DBMS_XPLAN

### 1. 실행통계 수집 힌트 추가한 SQL 수행.

```
SELECT  /*+ INDEX(A) GATHER_PLAN_STATISTICS */
        MAX(TRSP_RND_RT) MAX_RT
FROM    TRS_TRSP_AGMT_EQ_RT A
WHERE   EFF_TO_DT >= TO_DATE('20000101','YYYYMMDD')
AND     ROWNUM    <= 10000   ;
```

- ***Starts*** : *실제 수행 시 오퍼레이션을 시도한 수.*
- ***E-Rows*** : *예측 ROW 수.*
- ***A-Rows*** : *실제 수행 결과 ROW 수.*
- ***A-Time*** : *실제 수행 시간.(0.01초 단위)*
- ***Buffers*** : *실제 수행 시 엑세스 한 Memory 블록 버퍼 수.*
- ***Reads*** : *실제 수행 시 엑세스 한 Disk 블록 버퍼 수.*
- ***Writes*** : *실제 수행 시 Write 한 Disk 블록 수.*

### 2. 실행계획 및 실행정보 확인 [ DBMS_XPLAN.DISPLAY_CURSOR(NULL,NULL,'ADVANCED ALLSTATS LAST') ]

```
-------------------------------------------------------------------------------------------------------------------------------------------
| Id  | Operation                     | Name                 | Starts | E-Rows | A-Rows |   A-Time   | Buffers | Reads  |
-------------------------------------------------------------------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE               |                      |      1 |      1 |      1 |00:00:01.91 |     445 |    426 |
|*  2 |   COUNT STOPKEY               |                      |      1 |        |  10000 |00:00:00.19 |     445 |    426 |
|   3 |    TABLE ACCESS BY INDEX ROWID | TRS_TRSP_AGMT_EQ_RT  |      1 |   3877K|  10000 |00:00:00.19 |     445 |    426 |
|*  4 |     INDEX RANGE SCAN          | XAK2TRS_TRSP_AGMT_EQ_RT |      1 |   4075K|  10000 |00:00:00.02 |      59 |     59 |
-------------------------------------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
   2 - filter(ROWNUM<=10000)
   4 - access("EFF_TO_DT">=TO_DATE(' 2000-01-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND "EFF_TO_DT" IS NOT NULL)
```

실행계획은 위에서 아래방향으로, 안에서 밖으로,**JOIN**은 **PAIR**로, **JOIN**순서 **&** 방법은 각 **operation** 대로 확인

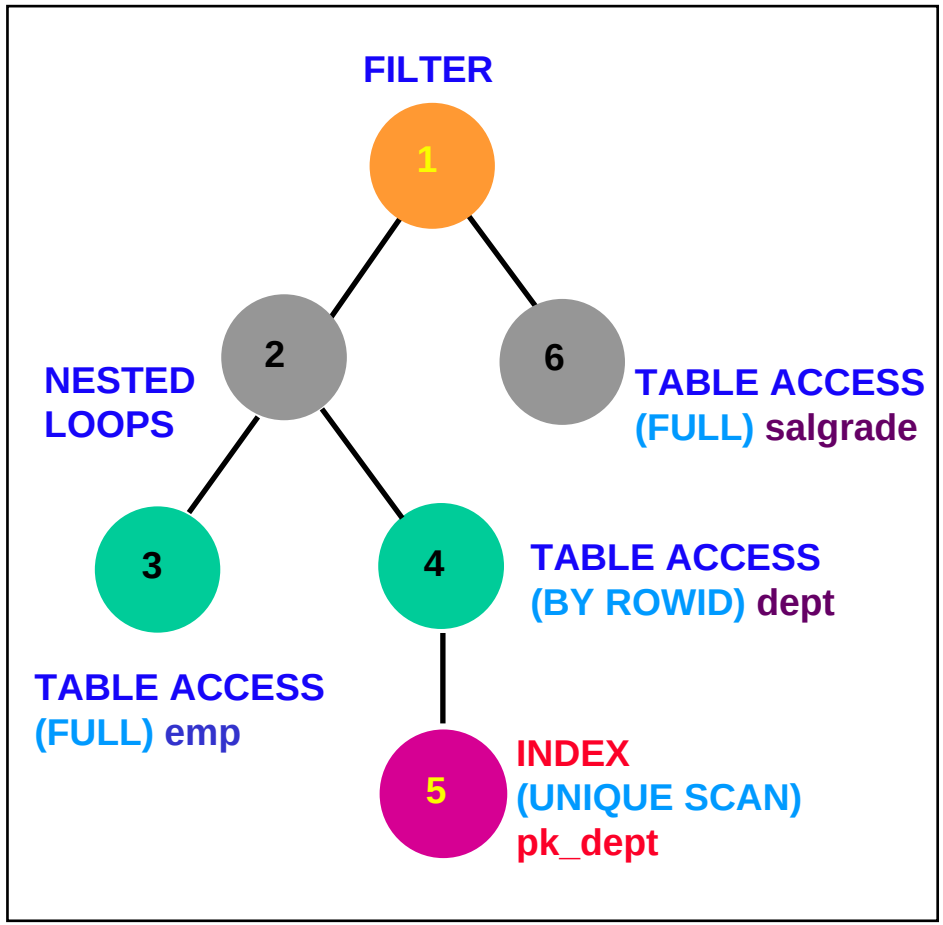
① 구문

```
SELECT ename, job, sal, dname
FROM     emp, dept
WHERE  emp.deptno = dept.deptno
AND  not exists
        (SELECT *
         FROM salgrade
         WHERE emp.sal BETWEEN losal AND hisal)
```

② 실행 계획

| | | | |
|---|---|---|---|
| 1 FILTER | | | |
| 2 | NESTED LOOPS | | |
| 3 | TABLE ACCESS FULL | EMP | |
| 4 | TABLE ACCESS BY ROWID | DEPT | |
| 5 | INDEX UNIQUE SCAN | PK_DEPT | |
| 6 | TABLE ACCESS | FULL | SALGRADE |

③ 실행 계획 분석 순서



| OPERATION | OPTION | 설 명 |
|---|---|---|
| **AGGREGATE** | | 그룹함수**(SUM, COUNT** 등**)** 사용하여 하나의 로우가 추출되도록 하는처리 |
| **AND-EQUAL** | | 인덱스 머지를 이용하는 경우 중복 제거**,** 단일 인덱스 컬럼을 사용하는 경우 |
| **CONNECT BY** | | **CONNECT BY**를 사용하여 트리구조로 전개 |
| **CONCATENATION** | | 단위 액세스에서 추출한 로우들의 합집합을 생성**(UNION-ALL)** |
| **COUNTING** | | 테이블의 로우 수를 센다**.** |
| **FILTER** | | 선택된 로우에 대해서 다른 집합에 대응되는 로우가 있다면 제거하는 작업 |
| **FIRST ROW** | | 조회 로우 중에 첫 번째 로우만 추출한다**.** |
| **FOR UPDATE** | | 선택된 로우에 **LOCK**을 지정한다**.** |
| **INDEX** | **UNIQUE**<br>**RANGE SCAN** | **UNIQUE** 인덱스를 사용**(**단 한 개의 로우를 추출**)**<br>**NON-UNIQUE**한 인덱스를 사용**(**한개 이상의 로우**)** |
| **INTERSECTION** | | 교집합의 로우를 추출한다**.(**같은 값이 없다**)** |
| **MINUS** | | **MINUS** 함수를 사용한다**.** |
| **MERGE JOIN** | | 먼저 자신의 조건만으로 액세스한 후 각각을 소트하여 머지해 가는 조인 |

| OPERATION | OPTION | 설 명 |
|---|---|---|
| **NESTED LOOPS** | | 드라이빙 테이블의 로우를 액세스한 후 그 결과를 이용해 다른 테이블을 연결하는 조인 |
| **REMOTE** | | 분산 데이타베이스에 있는 객체 추출 위해 데이타베이스 링크 사용하는 경우 |
| **SORT** | **AGGREGATE**<br>**UNIQUE**<br>**GROUP BY**<br>**JOIN**<br>**ORDER BY** | 그룹함수(**SUM, COUNT** 등)를 사용하여 하나의 로우가 추출되도록 하는 처리<br>같은 로우를 제거하기 위한 소트<br>액세스 결과를 **GROUP BY**하기 위한 소트<br>머지 조인을 하기 위한 소트<br>**ORDER BY**를 위한 소트 |
| **TABLE ACCESS** | **FULL**<br>**CLUSTER**<br>**HASH**<br>**BY ROWID** | 전체 테이블 스캔<br>클러스터 액세스<br>키값에 대한 해쉬 알고리즘을 사용<br>**ROWID**를 이용하여 테이블을 추출 |
| **UNION** | | 두 집합의 합집합을 구한다.(중복없음) 항상 전체 범위를 구한다. |
| **UNION ALL** | | 두 집합의 합집합을 구한다.(중복가능) **UNOIN**과 다르게 부분범위 처리를 한다. |
| **VIEW** | | 어떤 처리에 의해 생성되는 가상의 집합(뷰)에서 추출 |

## 실행계획과 조건 절

❑ **인덱스 드라이빙/검색 조건.(Driving/Search Condition)**

- 인덱스를 드라이빙/스캔 하는 양을 결정하는 조건.
- Inner 테이블인 경우 성능을 결정할 정도로 중요.
- 선행 컬럼 부터 연속된 조건만 가능.

❑ **인덱스 체크 조건.**

- 인덱스 드라이빙/검색 조건 이외의 인덱스 컬럼 절의 조건.

  인덱스의 엑세스 범위를 줄이지는 못하지만 테이블 엑세스량을 줄이는 역할.
- 인덱스 체크 조건은 Table로의 Random 엑세스를 줄여 준다.

❑ **테이블 체크 조건.**

- 드라이빙 인덱스 칼럼이 아닌 테이블의 모든 조건절의 상주 조건.

예) Select a.col1, a.col2, a.col3, a.col4, a.col5

```
From Tab1 a                  → Index Tab1_idx01 : col1 + col2 + col3
Where a.Col1 = 'aaa'         → Index Driving 조건.
And a.col3 = 'ccc'           → Index Check 조건.
And a.Col5 = 111             → Table Check 조건.
```

❑ **TABLE FULL SCAN 일 경우.**

- 모든 조건절이 테이블 체크 조건이 된다.(파티션은 파티션 키로 드라이빙)

## 실행계획 예

### Example 1

```
SELECT *
FROM   emp
WHERE  upper(ename) like 'PARK%'

ENAME_IDX : ENAME

Execution Plan
----------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE
  TABLE ACCESS (FULL) OF 'EMP'
```

### Example 2

```
SELECT *
FROM   emp
WHERE  upper(ename) like 'PARK%'

Execution Plan
----------------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE
  TABLE ACCESS (BY INDEX ROWID) OF 'EMP'
    INDEX (RANGE SCAN) OF 'ENAME_IDX' (NON-UNIQUE)
```

## 실행계획 예

Example 3

각각 **INDEX Column**이 **Where**의 조건이 미치는 영향은**?**

```
SELECT saledate, cust_code, description, item_id
FROM    sale
WHERE  saledate = :b1
AND      cust_code LIKE '%-BOM'
AND      NVL(end_date_active,sysdate+1) > SYSDATE ;
```

**sale_idx1 INDEX : saledate + cust_code + item_id**

```
Execution Plan
----------------------------------------------
SELECT STATEMENT
TABLE ACCESS BY INDEX ROWID SALE
  INDEX RANGE SCAN SALE_IDEX1
```

## 실행계획 예

Example 4

**Execution plan이 실행되는 순서는?**

**- NEST LOOP JOIN**

```
SELECT h.order_number, l.revenue_amount, l.ordered_quantity
FROM   sale h, item l
WHERE  h.saledate = :b1
AND    h.date_ordered > SYSDATE-30
AND    l.item_id = h.item_id ;
```

```
Plan
----------------------------------------------
SELECT STATEMENT
 NESTED LOOPS                                  ①
   TABLE ACCESS BY INDEX ROWID SALE            ②
     INDEX RANGE SCAN SALE_N1                  ③
   TABLE ACCESS BY INDEX ROWID ITEM            ④
     INDEX RANGE SCAN ITEM_N1                  ⑤
```

## 실행계획 예

Example 5

**Execution plan이 실행되는 순서는?**

**- HASH JOIN**

```
SELECT  *
FROM    sale a , item b
WHERE  a.item_id = b.item_id
AND      a.saledate = '20020303'
AND      b.unit_price = '1'
```

```
Execution Plan
----------------------------------------
SELECT STATEMENT Optimizer=CHOOSE
  HASH JOIN                                          ①
    TABLE ACCESS (FULL) OF 'ITEM '                   ②
    TABLE ACCESS (BY INDEX ROWID) OF 'SALE '         ③
      INDEX (RANGE SCAN) OF 'PK_SALE' (UNIQUE)       ④
```

## 실행계획 예

Example 6

**Execution plan이 실행되는 순서는?**

**- INLINE VIEW**

```
SELECT a.item_id ,b.sale_amt
FROM    item a ,
          (SELECT MAX( sale_amt ) sale_amt
           FROM   sale
           WHERE  saledate BETWEEN '20010101' AND '20020101'
           GROUP  BY saledate  ) b
WHERE  a.unit_price = b.sale_amt
```

```
Execution Plan
----------------------------------------------
SELECT STATEMENT
  NESTED LOOPS                                    ①
    VIEW                                          ②
      SORT (GROUP BY)                             ③
        TABLE ACCESS (BY INDEX ROWID) OF 'SALE'   ④
          INDEX (RANGE SCAN) OF 'PK_SALE' (UNIQUE)⑤
    TABLE ACCESS (FULL) OF 'ITEM'                 ⑥
```

## 실행계획 예

Example 7

**Execution plan**이 실행되는 순서는**?**

- **IN SUBQUERY**

```
SELECT item_id
FROM   item
WHERE  item_id IN (
                    SELECT item_id
                    FROM    sale
                    WHERE  saledate = '20021212' )
```

```
Execution Plan
----------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE
  NESTED LOOPS                                          ①
    VIEW OF 'VW_NSO_1 '                                 ②
      SORT (UNIQUE)                                     ③
        INDEX (RANGE SCAN) OF 'PK_SALE' (UNIQUE)        ④
    INDEX (UNIQUE SCAN) OF 'PK_ITEM' (UNIQUE)           ⑤
```

## 실행계획 예

Example 8

**Execution plan이 실행되는 순서는?**

**- NOT IN SUBQUERY**

```
SELECT 'A-2-2 ' ,
          NVL( pstn_brch_cd , ' ' ) ,
          NVL( jung_no , ' ' ) ,
          NVL( jung_seq_no , 0 ) ,
          NVL( firm_sym , ' ' )
FROM    tbjg12
WHERE  sangsil_dt = '99991231'
AND      pstn_brch_cd NOT IN (
                              SELECT brch_cd
                              FROM   tbtd10
                              WHERE  brch_adpt_yn = 'Y'  )
```

```
Execution Plan
----------------------------------------
SELECT STATEMENT - FIRST_ROWS- Cost Estimate:969
 FILTER                                                    ①
  TABLE ACCESS BY GLOBAL INDEX ROWID :JUNG                 ②
   INDEX RANGE SCAN :IX_JUNG_06(NU)(JUNG_SANGSIL_DT)       ③
  TABLE ACCESS BY INDEX ROWID :ZZT                         ④
   INDEX RANGE SCAN :PK_ZZT (U) (BRCH_CD,PSTN_TYPE)        ⑤
```

## 실행계획 예

Example 9

**- UPDATE**

```
UPDATE rm402 a
SET ( a.mat_cost , a.mat_amt) = (SELECT  b.in_cost ,  TRUNC( a.gy_wqty + a.gy_jqty)
                                 FROM    rm405 b
                                 WHERE  a.import_num = b.import_num
                                 AND       a.io_date LIKE '200209' ||'%'      )
WHERE  NVL( a.status , 'x' ) <> 'C'
AND      NVL( a.subl_flag , 'x' ) = 'Y'
AND      a.io_date LIKE '200209'||'%'
AND      EXISTS ( SELECT 'x'
                  FROM    rm405 b
                  WHERE  a.import_num = b.import_num
                  AND      a.io_date LIKE '200209' ||'%')
```

```
Execution Plan
----------------------------------------
UPDATE STATEMENT HINT=CHOOSE
  UPDATE TB_RM402                              ①
      FILTER                                   ②
         TABLE ACCESS BY INDEX ROWID RM402     ③
              INDEX RANGE SCAN PK_RM402        ④
          INDEX UNIQUE SCAN PK_RM405           ⑤
   FILTER                                      ⑥   ← SET절의 Sub-Query 처리
     TABLE ACCESS BY INDEX ROWID    RM405      ⑦
         INDEX UNIQUE SCAN PK_RM405            ⑧
```

## 실행계획 예

Example 10

**Execution plan**이 실행되는 순서는?

```
Execution Plan
----------------------------------------
SELECT STATEMENT - CHOOSE
 SORT GROUP BY                                          ①
  NESTED LOOPS                                          ②
   VIEW US_SWJP.(1)                                     ③
    UNION-ALL                                           ④
     SORT GROUP BY                                      ⑤
      NESTED LOOPS                                      ⑥
       TABLE ACCESS FULL :US_SWJP.TB_CM206(3)           ⑦
       TABLE ACCESS BY INDEX ROWID :US_SWJP.TB_DC530(2) ⑧
         INDEX RANGE SCAN :US_SWJP.IX_DC530_01(NU)      ⑨
     SORT GROUP BY                                      ⑩
      NESTED LOOPS                                      ⑪
       TABLE ACCESS FULL :US_SWJP.TB_CM206(5)           ⑫
       TABLE ACCESS BY INDEX ROWID :US_SWJP.TB_DC400(4) ⑬
         INDEX RANGE SCAN :US_SWJP.PK_DC400 (U)         ⑭
   TABLE ACCESS BY INDEX ROWID :US_SWJP.TB_CM110(6)     ⑮
      INDEX UNIQUE SCAN :US_SWJP.PK_CM110 (U) (GRAS)                    (16)
```

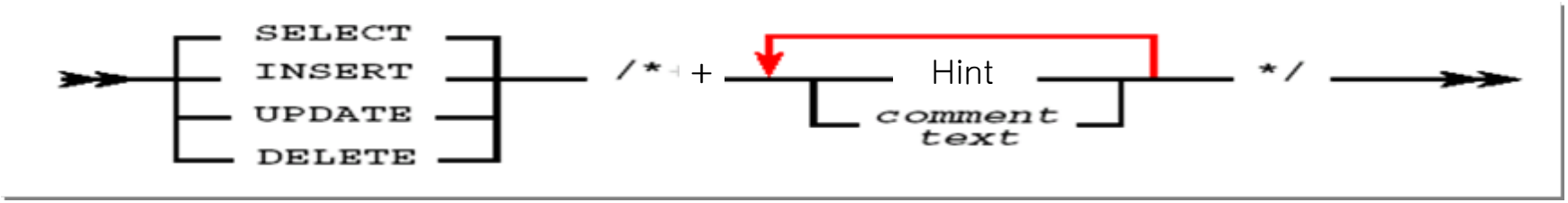
# Optimizer Hint

## Hint의 정의

- Optimizer가 항상 최적의 Execution Plan을 생성하지는 않음.
- CBO가 주어진 쿼리에 대해서 최적의 Plan을 생성하는데 도움을 제공하는 키워드 (가이드이며 명령이 아니다).
- SQL 개발자가 액세스되는 User Data에 대해서 Optimizer보다 더 잘 안다고 가정.
- Hint는 Query의 결과에 영향을 주지 않음.
- 잘못 사용된 Hint는 Optimizer에 의해 무시됨 (문법 오류, Query 결과에 영향을 주는 Hint 등)
- Hint는 RULE, APPEND, CURSOR_SHARING_EXACT 를 제외하고는 항상 CBO 를 호출하며, FIRST_ROWS를 제외하고 모두 ALL_ROWS로 수행.

## Hint의 사용 규칙

- SQL 블록의 첫 키워드 바로 뒤에 입력.
- 각 블록에서 첫번째 Hint 주석만 항상 인식, 하나의 Hint 주석은 여러 개의 Hint 포함 가능.
- Hint는 해당 블록에만 적용.
- 문장에 alias를 사용하는 경우 힌트는 그 alias를 참조해야 함.



## Hint의 파싱

- 힌트는 앞에서 하나의 쿼리 블록에 여러 개 기술 가능, 단 첫번째만 힌트로 인식 나머지는 주석 처리.
  **(** 주 처리 와 검증시 등의 개별 목적에 따라 사용 가능하다. **)**
- 힌트절 안에서 힌트 구문 이외는 다 주석으로 인식.
  **(** 구문이 힌트절 인지의 여부는 옵티마이져가 판단 **-** 철자 오류에 대한 처리 여부 **)**
- 힌트의 종류에 따라 이후 힌트의 파싱 및 적용 범위 및 여부 결정
  - 철자 오류, 구문 오류시 해당 힌트만, 또는 이후의 모든 힌트 무시가 각각 발생 한다.
- 의도적 무시
  - 전체 범위 처리 **SQL**에서 **FIRST_ROWS**와 같은 힌트는 무시.
  - 개별 쿼리 블록마다 **OPTIMIZER_GOAL**이 다른 경우도 무시.
  - 서로 상반되는 성격의 힌트 지정시 무시.

## Hint의 종류

- **Hints for Optimization Approaches and Goals**
- **Hints for Access Methods**
- **Hints for Join Orders**
- **Hints for Join Operations**
- **Hints for Parallel Execution**

## 자주 사용하는 Hints

| Hint | USE |
|---|---|
| /*+ ALL_ROWS */ | cost-based optimizer에서 전체 응답시간이 가장 적은 plan 선택(DW) |
| /*+ FIRST_ROWS(100) */ | cost-based optimizer에서 첫번째 n row가 가장 빨리 나오는 plan으로 선택(OLTP) |
| /*+ RULE */ | rule-based optimization로 plan 작성 |
| /*+ **FULL**(table) */ | index 유무에 상관없이 full table scan 선택 |
| /*+ HASH_AJ(table) */ | NOT IN subquery를 hash antijoin으로 변환 |
| /*+ **INDEX**(table index) */ | 특정 table의 특정 index를 순방향으로 사용 |
| **/*+ INDEX_DESC(table index) */** | 특정 table의 특정 index를 역방향으로 사용 |
| /*+ INDEX_FFS(table index) */ | index만으로 구성된 sql에서 fast full index scan 사용 |
| /*+ **ORDERED** */ | FROM 절에 나온 순서대로 join 순서 조정 |
| /*+ **USE_HASH** (table) */ | hash join 사용 |
| /*+ **USE_NL** (table) */ | nested-loops join 사용 |
| /*+ APPEND */ | INSERT mode에서만 사용되며 기존의 HWM 밑의 free space를 사용하지 않고 HWM위에 append 함 |
| /*+ PARALLEL(table degree) */ | table의 parallel degree 지정 |

## 옵티마이저 모드에 대한 Hints

Optimizer 모드에 관한 힌트는 rule-based optimizer와 cost-based optimizer 중 sql문 레벨에서 선택할 수 있도록 하고 cost-based optimizer인 경우 최고의 throughput과 최고의 응답 시간중 선택을 할 수 있도록 한다.

**(1) RULE**

sql문 레벨에서 rule-based optimizer를 선택하고 optimizer로 하여금 rule 힌트 외에 다른 모든 힌트는 무시하도록 한다.

```
SELECT /*+ RULE */ empno, ename, sal, job  FROM emp WHERE empno = 7566;
```

**(2) CHOOSE**

Choose 힌트가 주어졌을 때 sql문상의 테이블들 중 적어도 하나의 테이블에 대한 통계정보가 dictionary에 존재하면 optimizer는 cost-based optimizer를 사용하고 최고의 throughput을 낸다.

```
SELECT /*+ CHOOSE */ empno, ename, sal, job FROM emp WHERE empno = 7566;
```

**(3) ALL_ROWS**

ALL_ROWS 힌트는 최고의 throughput(최소의 자원 소비)을 목표로 sql문을 optimize한다.

```
SELECT /*+ ALL_ROWS */ empno, ename, sal, job FROM emp WHERE empno = 7566;
```

**(4) FIRST_ROWS(100)**

FIRST_ROWS 힌트는 최고의 응답시간을 목표로 sql문을 optimize한다. 이것은 곧 First n row를 빨리 return하는 것이 목적이고 최소의 자원을 사용해서 이러한 작업이 이루이지도록 하는것이다.

```
SELECT /*+ FIRST_ROWS(100) */ empno, ename, sal, job FROM emp WHERE empno = 7566;
```

## ACCESS PATH에 대한 Hints

아래의 힌트를 지정함으로서 인덱스나 클러스터등의 존재여부를 기준으로 가능한 access path인 경우에만 해당 access path를 선택한다. 힌트에 지정된 access path가 불가능하면 optimizer는 그 힌트을 무시한다

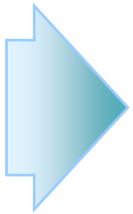
| 힌트의 종류 | 힌트사용 방법 | 힌트의 내용 |
|---|---|---|
| FULL | /*+ FULL (테이블명) */ | 지정된 테이블을 full table scan하도록 한다. |
| ROWID | /*+ ROWID (테이블명) */ | 지정된 테이블을 rowid로 scan할 수 있도록 한다. |
| CLUSTER | /*+ CLUSTER (테이블명) */ | 지정된 테이블을 access하기 위해 cluster scan을 수행한다. 이 힌트는 cluster된 object에 대해서만 적용된다. |
| HASH | /*+ HASH (테이블명) */ | 지정된 테이블을 access하기 위해 hash scan을 수행한다. 이 힌트는 cluster에 저장된 테이블에 대해서만 적용된다. |
| HASH_AJ | /*+ HASH_AJ */ | 지정된 테이블을 access하기 위해 NOT IN subquery를 hash anti 조인으로 변형한다. |
| HASH_SJ | /*+ HASH_SJ (테이블명) */ | 지정된 테이블을 access하기 위해 상호 관련이 있는 EXISTS subquery를 hash semi 조인으로 변형한다. |
| INDEX | /*+ INDEX (테이블명,인덱스명1,인덱스명2...) */ | 지정된 테이블을 인덱스 스캔할 수 있도록 한다. |
| INDEX_ASC | /*+INDEX_ASC (테이블명, 인덱스명) */ | 지정된 테이블에 대해 인덱스 스캔할 수 있도록 한다. 오름차순으로 인덱스 엔트리를 scan한다. |
| INDEX_DESC | /*+INDEX_DESC (테이블명, 인덱스명) */ | 지정된 테이블에 대해 인덱스 스캔할 수 있도록 한다.내림차순으로 인덱스 엔트리를 scan한다. |
| INDEX_FFS | /*+INDEX_FFS (테이블명, 인덱스명) */ | 이 힌트는 full table scan보다는 fast full index scan을 하도록 한다. |
| MERGE_AJ | /*+MERGE_AJ */ | 지정된 테이블을 access 하기위해 NOT IN subquery를 merge anti 조인으로 변형한다. |
| MERGE_SJ | /*+MERGE_SJ */ | 지정된 테이블을 access 하기위해 상호관련있는 EXISTS subquery를 merge semi 조인으로 변형한다. |

## JOIN 순서에 대한 Hints

### (1) ORDERED

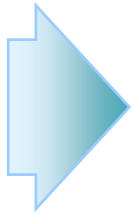
이 힌트는 From절에 나타난 테이블 순서로 조인할 수 있도록 한다. 아래의 sql문을 보면 tab1테이블 첫 outer 테이블이 되어 tab2를 한다.

```
SELECT /*+ ORDERED */ count(*)
FROM   TB_GWB04 C,
       TB_GWB05 A
WHERE A.CAPP_REQ_NO = C.CAPP_REQ_NO
  AND A.PROC_STAT = '0104'
  AND C.REQ_DT  LIKE '200503%'
```



```
SELECT STATEMENT Optimizer=CHOOSE
 SORT (AGGREGATE)
  HASH JOIN
      TABLE ACCESS (BY INDEX ROWID) OF 'TB_GWB04'
        INDEX (RANGE SCAN) OF 'IX_GWB04_03' (NON-UNIQUE) (
      TABLE ACCESS (FULL) OF 'TB_GWB05'
```

```
SELECT /*+ ORDERED */ count(*)
FROM  TB_GWB05 A,
      TB_GWB04 C
WHERE A.CAPP_REQ_NO = C.CAPP_REQ_NO
  AND A.PROC_STAT = '0104'
  AND C.REQ_DT  LIKE '200503%'
```



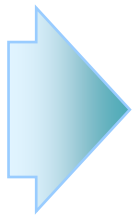
```
SELECT STATEMENT Optimizer=CHOOSE
  SORT (AGGREGATE)
    HASH JOIN
      TABLE ACCESS (FULL) OF 'TB_GWB05'
      TABLE ACCESS (BY INDEX ROWID) OF 'TB_GWB04'
        INDEX (RANGE SCAN) OF 'IX_GWB04_03' (NON-UNIQUE)
```

## JOIN 연산에 대한 Hints

### (1) USE_NL : /*+USE_NL(테이블명,테이블명,..) */

이 힌트는 Oracle로 하여금 지정된 각 테이블을 inner 테이블로 사용하여 다른 row source에 nested loop 조인이 이루어지도록 한다. 아래의 accounts 테이블과 customers 테이블을 조인하는 sql문이 있다.

```
SELECT /*+ ORDERED USE_NL(A C)  */
count(*)
FROM  TB_GWB05 A,
      TB_GWB04 C
WHERE A.CAPP_REQ_NO = C.CAPP_REQ_NO
  AND A.PROC_STAT = '0104'
  AND C.REQ_DT  LIKE '200503%'
```

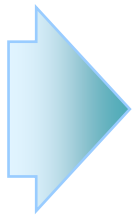


```
SELECT STATEMENT Optimizer=CHOOSE
  SORT (AGGREGATE)
    NESTED LOOPS
      TABLE ACCESS (FULL) OF 'TB_GWB05'
      TABLE ACCESS (BY INDEX ROWID) OF 'TB_GWB04'
        INDEX (UNIQUE SCAN) OF 'IX_GWB04_PK' (UNIQUE)
```

### (2) USE_MERGE : /*+USE_MERGE(테이블명,테이블명,..) */

이 힌트는 Oracle로 하여금 지정된 각 테이블을 다른 row source와 sort merge 조인할 수 있도록 한다. 지정되는 테이블은 앞선 조인 결과 row source와 sort merge조인할 테이블이 된다.

```
SELECT /*+ ORDERED USE_MERGE(A C)  */
count(*)
FROM  TB_GWB05 A,
      TB_GWB04 C
WHERE A.CAPP_REQ_NO = C.CAPP_REQ_NO
  AND A.PROC_STAT = '0104'
  AND C.REQ_DT  LIKE '200503%'
```



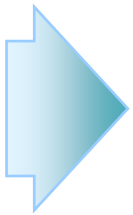
```
SELECT STATEMENT Optimizer=CHOOSE
  SORT (AGGREGATE)
    MERGE JOIN
      SORT (JOIN)
        TABLE ACCESS (FULL) OF 'TB_GWB05'
      SORT (JOIN)
        TABLE ACCESS (BY INDEX ROWID) OF 'TB_GWB04'
          INDEX (RANGE SCAN) OF 'IX_GWB04_03'
```

## JOIN 연산에 대한 Hints

### (3) USE_HASH : /*+USE_HASH(테이블명,테이블명,..) */

이 힌트는 oracle로 하여금 지정된 각 테이블을 다른 row source와 hash 조인할 수 있도록 한다. 지정되는 테이블은 앞선 조인 결과 row source와 hash 조인할 테이블이 된다.

```
SELECT /*+ ORDERED USE_HASH(A C) */
       count(*)
FROM   TB_GWB05 A,
       TB_GWB04 C
WHERE  A.CAPP_REQ_NO = C.CAPP_REQ_NO
  AND  A.PROC_STAT = '0104'
  AND  C.REQ_DT  LIKE '200503%'
```



```
SELECT STATEMENT Optimizer=CHOOSE
  SORT (AGGREGATE)
    HASH JOIN
      TABLE ACCESS (FULL) OF 'TB_GWB05'
      TABLE ACCESS (BY INDEX ROWID) OF 'TB_GWB04'
        INDEX (RANGE SCAN) OF 'IX_GWB04_03' (NON-UNIQUE)
```

# 4. 조인

## Nested Loop Join

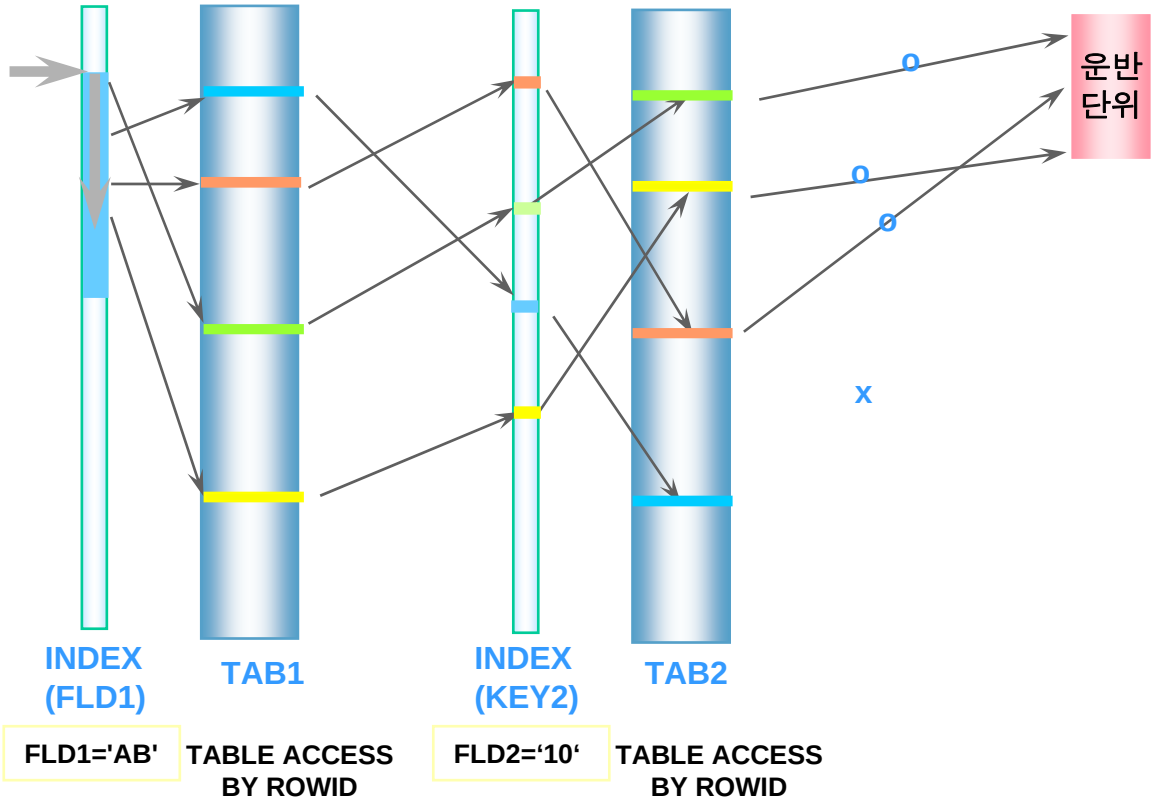
**① Join 방식**

Driving Table에서 Row를 추출한 후 그 결과를 다른 테이블에 연결하여 Join

**② 특징**

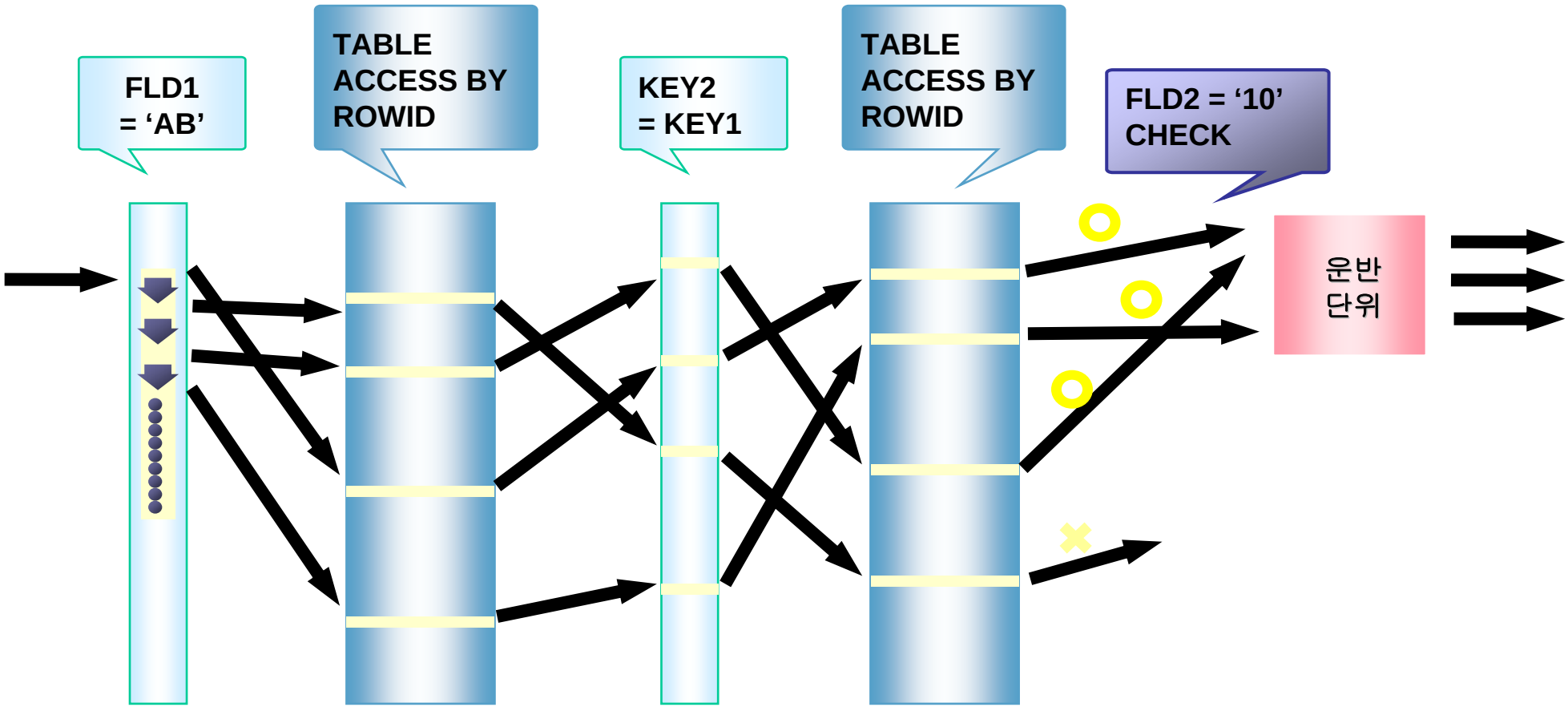
- 순차적
  : 부분범위처리 가능
- 종속적
  : 먼저 처리되는 테이블의 처리범위에 따라 처리량 결정
- 랜덤(Random) 액세스 위주
- 연결고리 상태에 따라 영향이 큼
- 주로 좁은 범위 처리에 유리

```
SELECT a.FLD1, ..., b.FLD1,...
FROM   TAB1 a,  TAB2 b
WHERE  a.KEY1 = b.KEY2   AND   a.FLD1 = 'AB'  AND   b.FLD2 = '10'
```



## Nested Loop Join

```
SELECT A.FLD1, …, B.COL1….
  FROM TAB1 A, TAB2 B
 WHERE A.KEY1 = B.KEY2
   AND A.FLD1 = 'AB'
   AND B.FLD2 = '10'
```



## Sort Merge Join

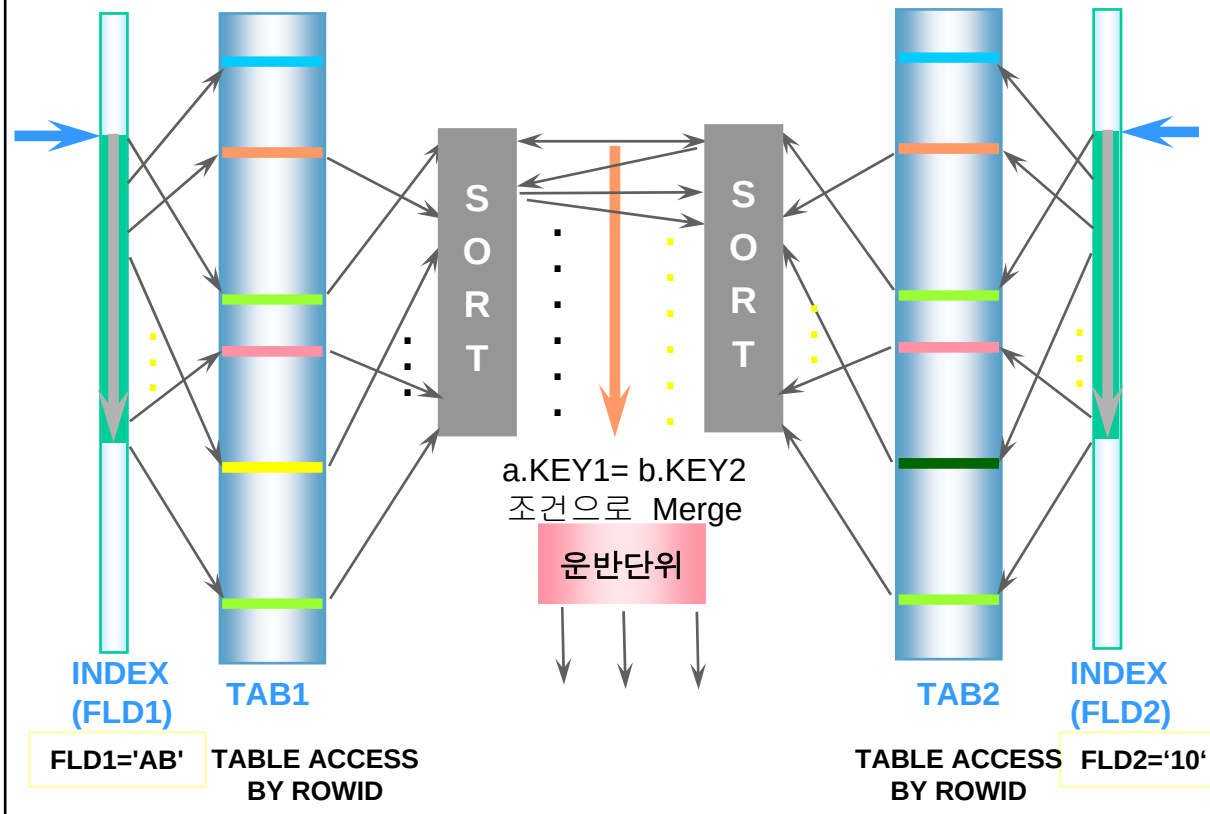
**① Join 방식**

양쪽 테이블의 처리범위를 각각 억세스하여
정렬한 결과를 차례로 스캔하면서
연결고리의 조건을 만족하는지를
머지(Merge)해 가는 방식

**② 특징**

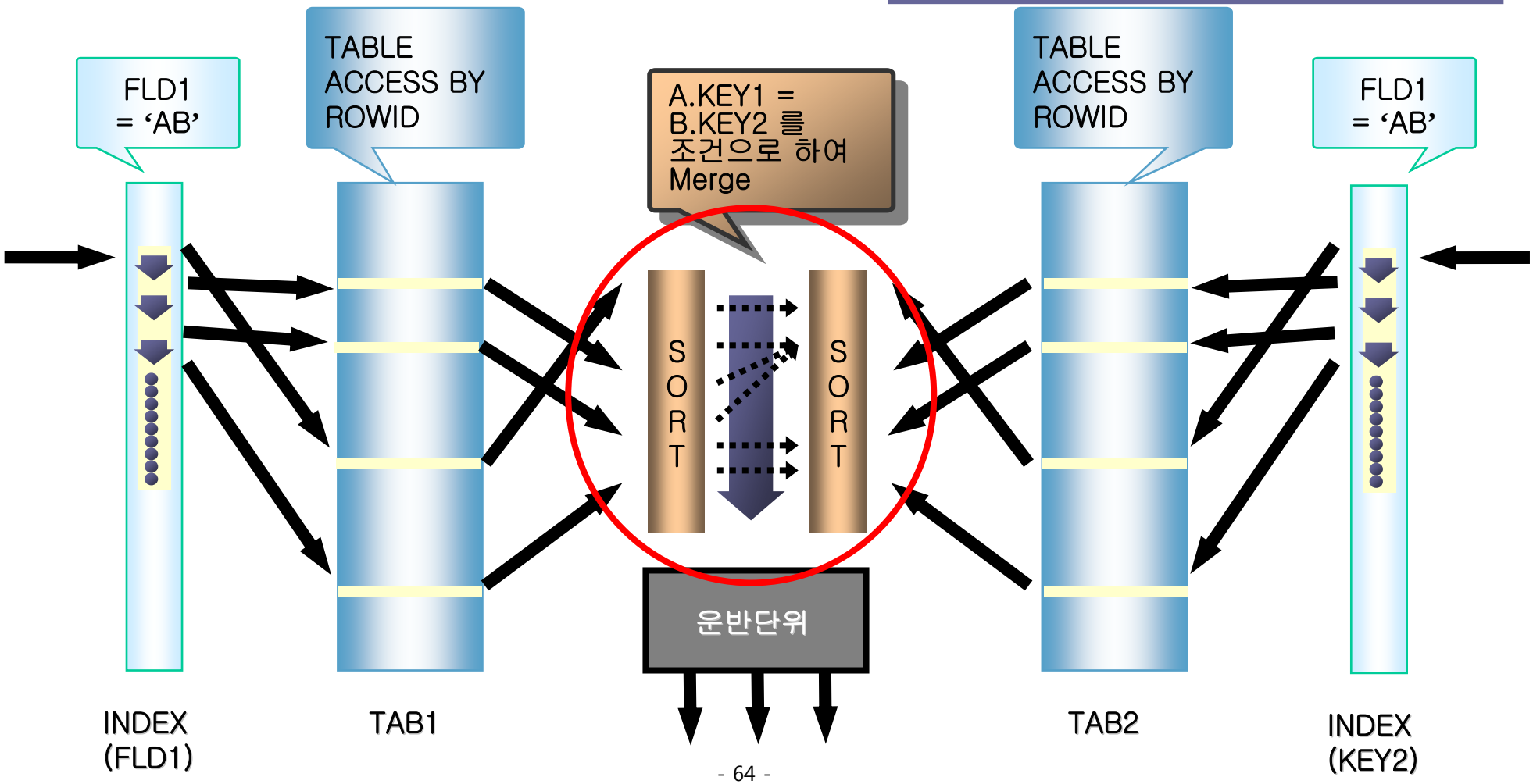
- 동시적
  : 무조건 전체범위처리
- 독립적
  : 자기의 처리범위만으로 처리량 결정
- 스캔(Scan) 액세스 위주
- 연결고리 상태에 영향이 없음
- 주로 넓은범위 처리에 유리

```
SELECT /*+ use_merge(a b) */ a.FLD1, ..., b.FLD1,...
FROM    TAB1 a,  TAB2 b
WHERE  a.KEY1 = b.KEY2   AND   a.FLD1 = 'AB'  AND   b.FLD2 = '10'
```

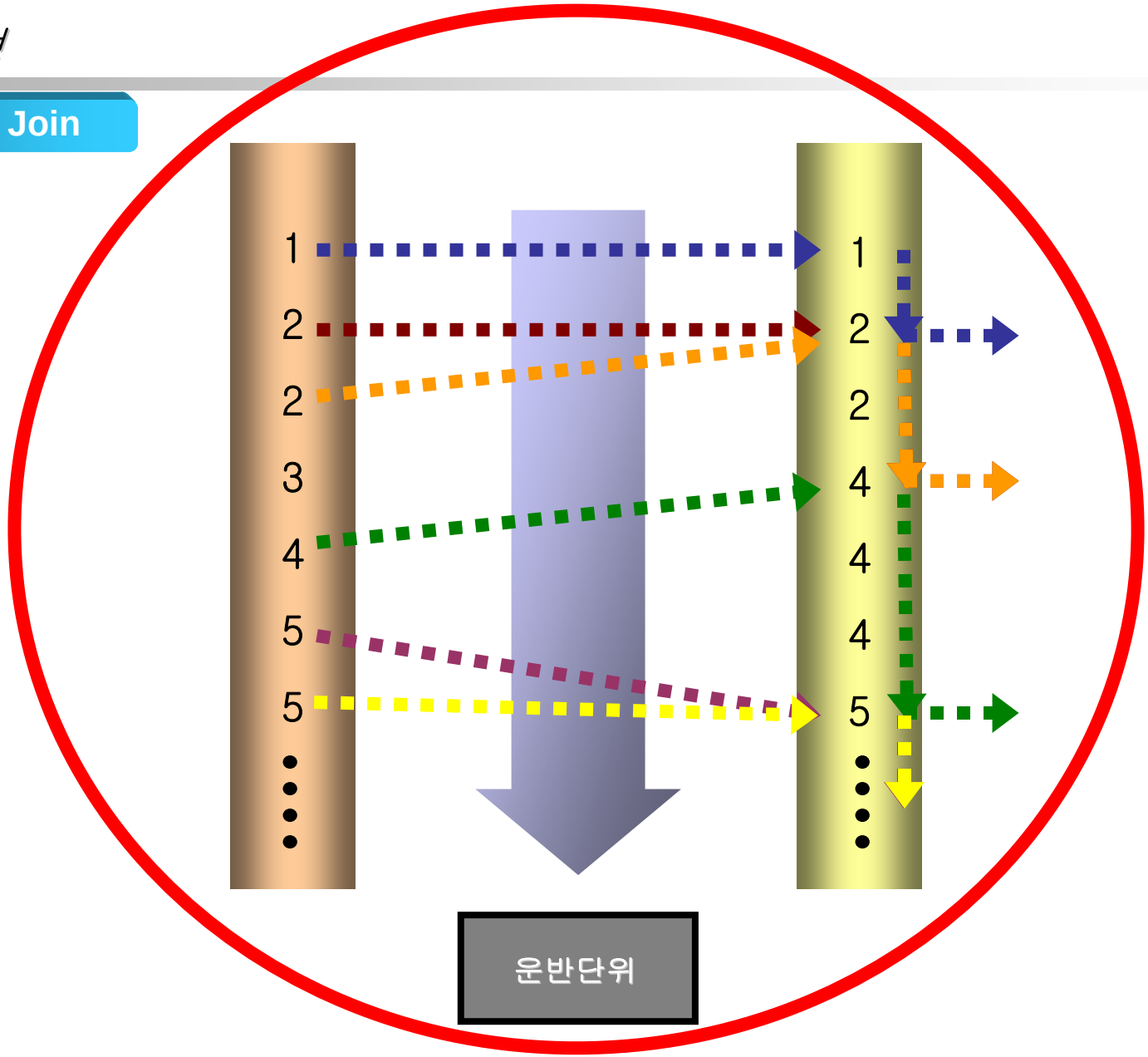


## Sort Merge Join

```
SELECT A.FLD1, ..., B.COL1....
  FROM TAB1 A, TAB2 B
 WHERE A.KEY1 = B.KEY2
   AND A.FLD1 = 'AB'
   AND B.FLD2 = '10'
```



## Sort Merge Join




Sort Merge Join에서의 Merge Operation

## Hash Join

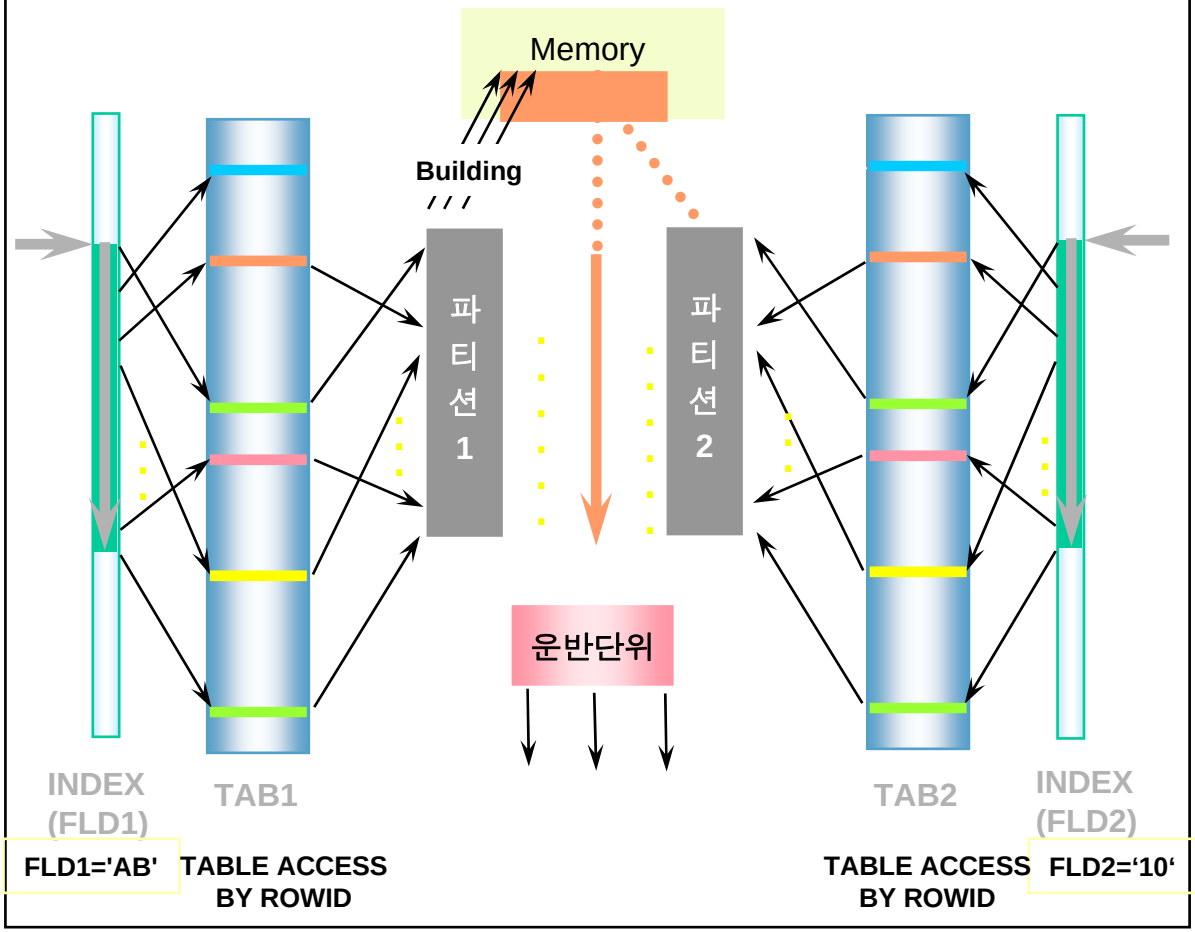
**① Join 방식**

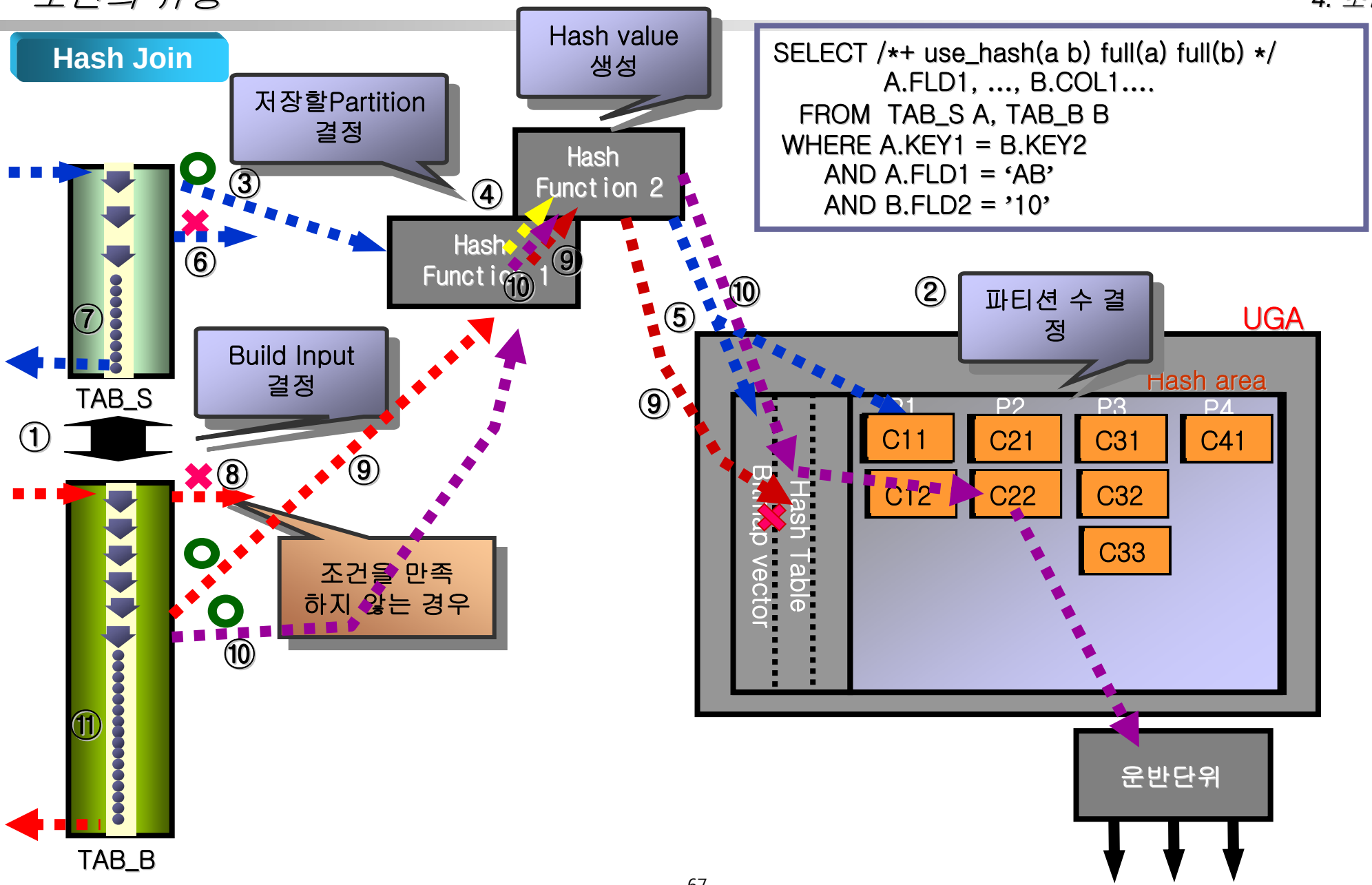
크기가 작은 파티션을 메모리에 로딩(Building)하여 Hash Table 생성 후, 나머지 파티션의 Row를 읽어 Hash Table상 대응되는 로우 체크

**② 특징**

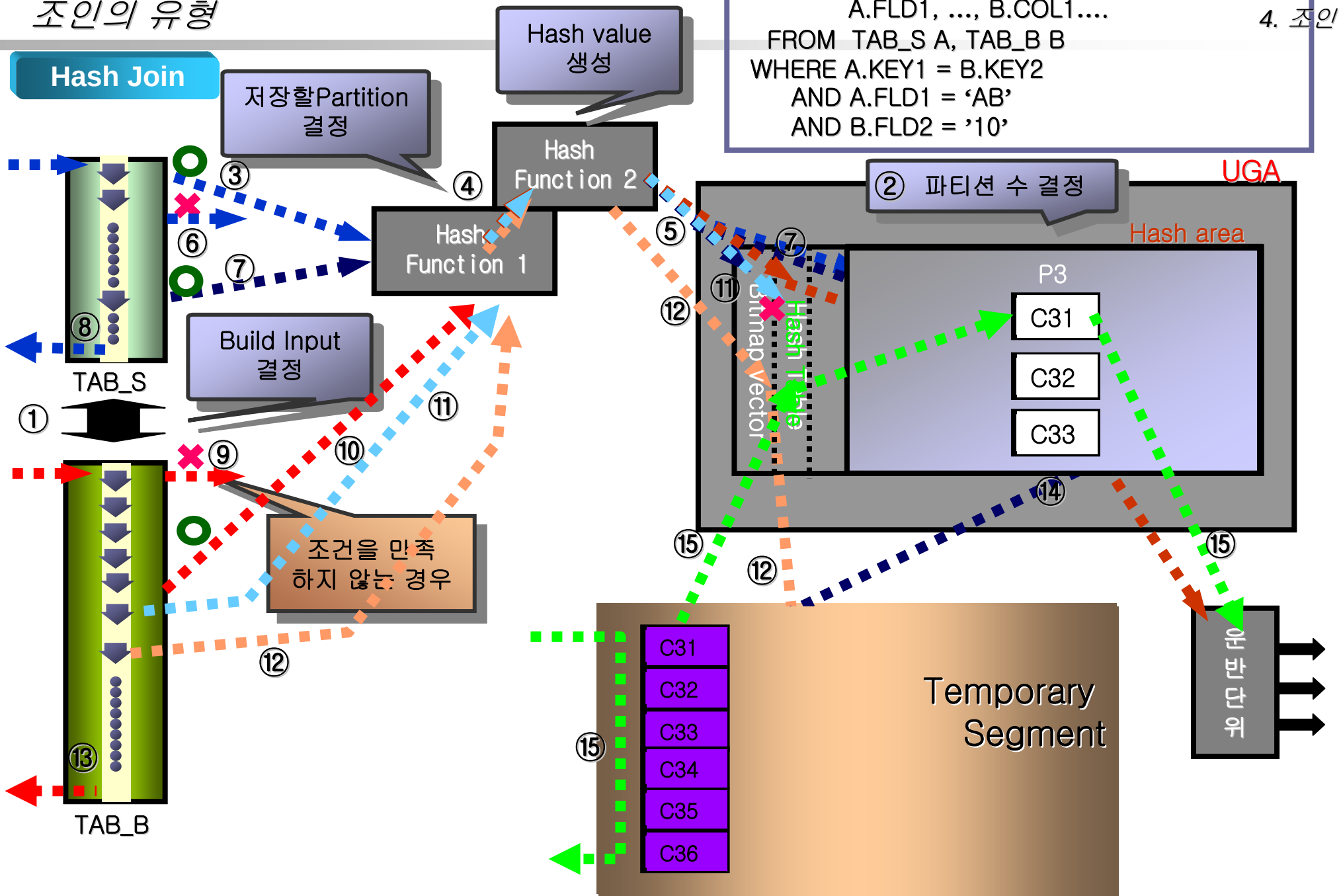
- 독립적
  : 자기의 처리범위만으로 처리량 결정
- 반 부분범위 처리
  :Hash Function을 이용하여 매핑하는 후행 테이블은 전체범위 처리 수행
- 메모리 영역만으로 Hash Table 생성시 최적의 효과 가능하므로 적은 테이블이 선행테이블로 선택됨
- Hash Function을 이용하므로 결과값 정렬 보장 받을 수 없음

```
SELECT /*+ use_hash(a b) */ a.FLD1, ..., b.FLD1,...
FROM    TAB1 a,  TAB2 b
WHERE  a.KEY1 = b.KEY2   AND   a.FLD1 = 'AB'  AND   b.FLD2 = '10'
```




Hash Join
Hash value 생성
저장할Partition 결정
Hash Function 2
Hash Function 1
SELECT /*+ use_hash(a b) full(a) full(b) */
A.FLD1, ..., B.COL1....
FROM TAB_S A, TAB_B B
WHERE A.KEY1 = B.KEY2
AND A.FLD1 = 'AB'
AND B.FLD2 = '10'
② 파티션 수 결정
UGA
Hash area
P1
P2
P3
P4
C11
C21
C31
C41
C12
C22
C32
C33
Hash Table
Bitmap vector
Build Input 결정
조건을 만족 하지 않는 경우
TAB_S
TAB_B
운반단위
① ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩ ⑪

```
SELECT /*+ use_hash(a b) full(a) full(b) */
       A.FLD1, ..., B.COL1....
  FROM  TAB_S A, TAB_B B
 WHERE A.KEY1 = B.KEY2
   AND A.FLD1 = 'AB'
   AND B.FLD2 = '10'
```



## Outer Join

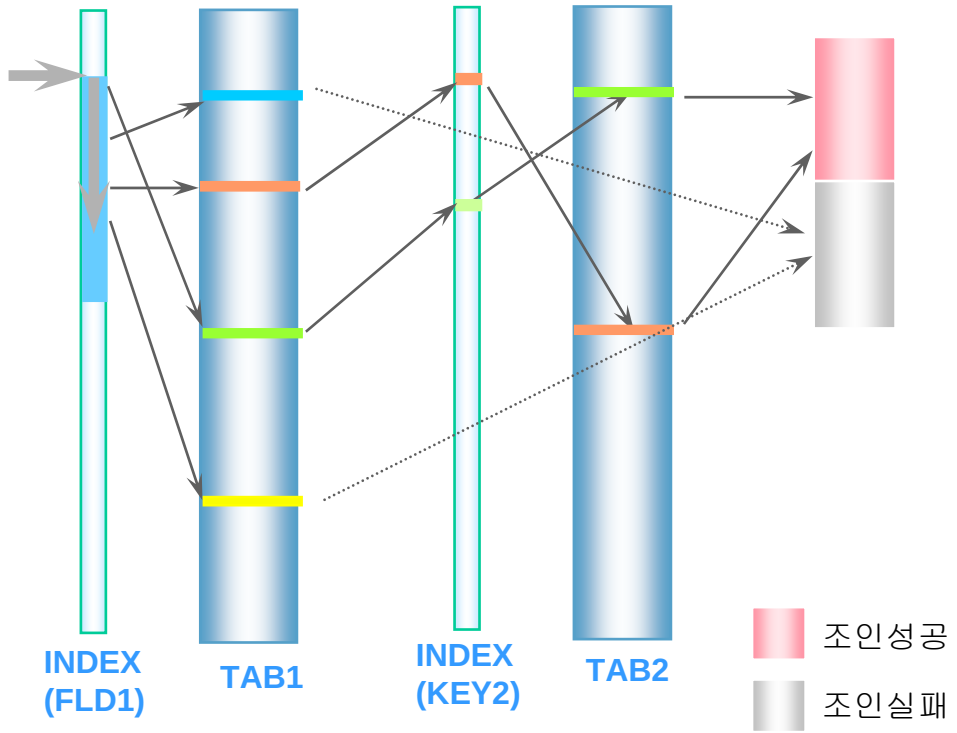
**① Join 방식**

조인조건에 만족되지 않더라도 결과에 포함시키기 위한 특별한 조인기법

**② 특징**

- 조인순서가 미리 정해지므로 조인순서 이용한 튜닝이 불가
- ORDERED 힌트가 outer join 순서에 위배된다면 무시됨
- (+) 기호를 이용하여 IN, OR의 연산자를 이용하여 비교 불가능(Inline View 이용)
- (+) 기호를 이용하여 Subquery와 불가능(IS NULL OR 조건과 같이 비교)

```
SELECT a.FLD1, ..., b.FLD1,...
FROM     TAB1 a,  TAB2 b
WHERE  a.KEY1 = b.KEY2(+)
     AND   a.FLD1 = 'AB'  AND   b.FLD2 (+)= '10'
```



## Join 유형 비교 평가

| | Nested Loop | Hash | Sort Merge |
|---|---|---|---|
| 처리 방식 | 순차적( 완전 부분범위 ) | 반 부분범위 | 동시적(전체범위처리) |
| **Access** 방식 | Random Access | Hash Function | Scan 방식 |
| 연결 고리 | 절대 영향 | 영향 없음 | 영향 없음 |
| **Join** 방향 | 영향 큼 | 영향 있음 (Hash Table 구성) | 영향 없음 |
| 사용 **resource** | BUFFER CACHE | PGA | PGA |
| 처리량 | 좁은 범위에 유리 | 넓은 범위에 유리 | 넓은 범위에 유리 |
| 주요 **Check** 요소 | 연결 고리 상태 및 처리량 | Hash_area_size<br>Hash Table size | Sort_area_size<br>각 Table 의 Sort 량<br>(Temp 사용량) |

- **Nested Loop Join**은 대용량의 **Data**에서 **Random Access** 매우 취약.
- **Hash Join**은 **Sort Merge Join**에 비해서 거의 모든 면에서 유리.

# 5. SQL 활용

# Scalar SubQuery Expression

## 기본 개념

- Scalar Subquery 는 쿼리 수식으로부터 유도된 스칼라 값을 지정하기 위해 사용.
- Scalar Subquery 수행 결과는 오직 하나의 값만 반환.
  반환되는 값의 데이터 형은 서브 쿼리에서 선택되는 데이터 형과 일치 해야 한다.
- Scalar Subquery 결과 값이 0 row 이면 null Value로 Return된다.
- Oracle9i 에서 스칼라 서브 쿼리는 유효한 수식이 쓰일 수 있는 모든 곳에서 사용 가능.(Oracle8i 부터 사용가능)
  - GROUP BY를 제외한 모든 SELECT 절
  - INSERT 문의 VALUES 절
  - UPDATE 문의 SET 절 및 WHERE 절
  - DECODE 및 CASE의 조건 또는 수식
- 기본 Syntax

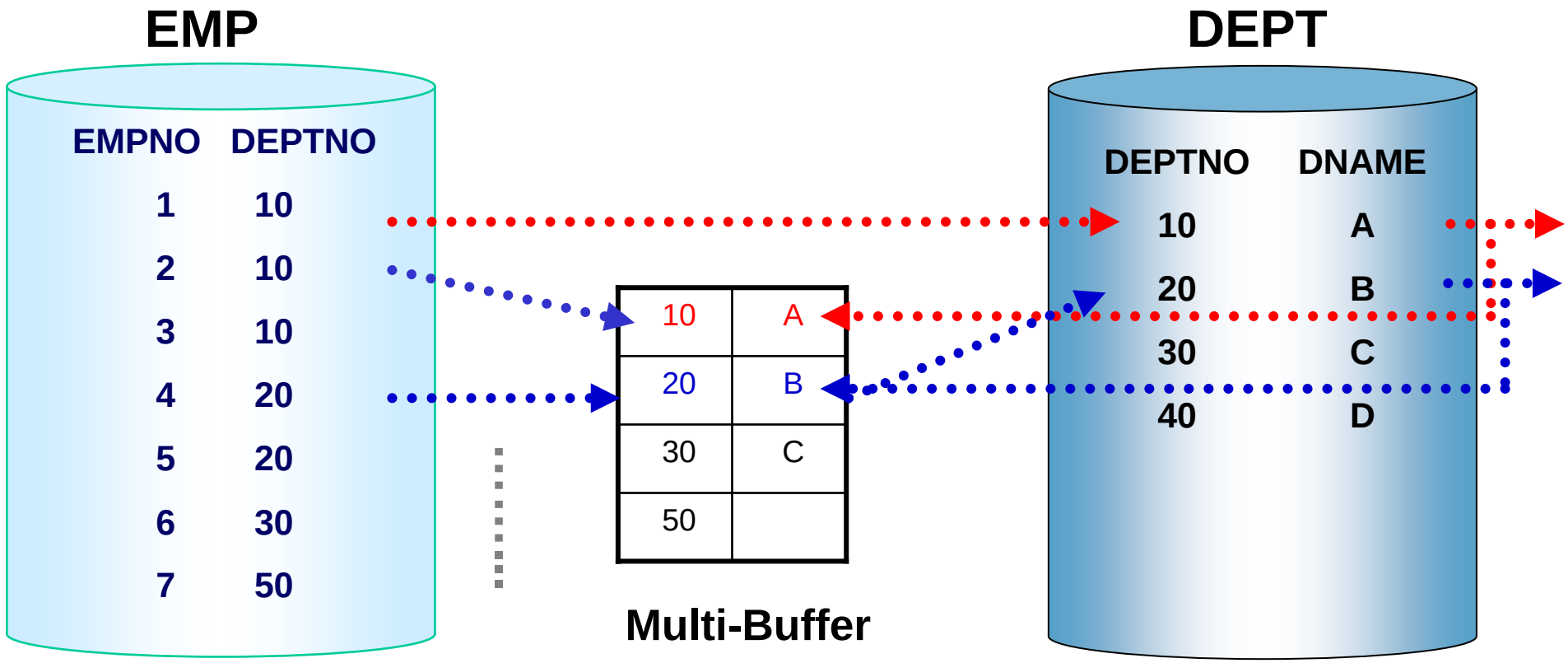
```
SELECT  EMPNO,
        (SELECT  DNAME
         FROM    DEPT B
         WHERE   B.DEPTNO = A.DEPTNO) DNAME
FROM EMP A
```

**활용 적용 기준**

**M:1 Degree Gap**이 많이 차이나는 부모 측 집합과 연결시 **1 SQL Execution** 사용개발환경고려

수행원리

```
SELECT  EMPNO,
        (SELECT  DNAME
         FROM    DEPT B
         WHERE   B.DEPTNO = A.DEPTNO) DNAME
FROM EMP A
```



## Scalar SubQuery Expression의 성능

### Nested-loop join 방식 일때

```
select count(slp_stlcd)
from (
select /*+ full(a) use_nl(a b) */ slp_stlcd
from   maccount.acm201 b , maccount.acm203 a
where  b.finac_yymm = a.finac_yymm
and    b.finac_dy   = a.finac_dy
and    b.regdp_cd   = a.regdp_cd
and    b.slp_srno   = a.slp_srno
and    rownum <= 100000)

call       count       cpu    elapsed       disk      query    current        rows
-----  ---------- ---------- ----------  ---------- ---------- ----------  ----------
Parse          1       0.01       0.01          0          0          0           0
Execute        1       0.00       0.00          0          0          0           0
Fetch          2       2.27       2.28          0     402501         12           1
-----  ---------- ---------- ----------  ---------- ---------- ----------  ----------
total          4       2.28       2.29          0     402501         12           1

Rows     Row Source Operation
-------------------------------------------------
      1  SORT AGGREGATE
 100000   VIEW
 100000    COUNT STOPKEY
 100000     NESTED LOOPS
 100000      TABLE ACCESS FULL ACM203
 100000      TABLE ACCESS BY INDEX ROWID ACM201
 100000       INDEX UNIQUE SCAN (object id 33200)
```

전표(ACM201)
# 전표년월
# 전표일
# 전표발행부서
# 전표번호
* 전표종류
……

전표내역(ACM203)
# 항번
* 계정
…..

## Scalar SubQuery Expression의 성능

**Function 처리 방식 일때**

```
select count(slp_stype)
from (
select f1(finac_yymm, finac_dy, regdp_cd, slp_srno) slp_stype
from   maccount.acm203
where  rownum <= 100000)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2     18.74      21.86          0       2501         12           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4     18.75      21.86          0       2501         12           1

Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 61  (ADMACCOUNT)

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE
 100000   VIEW
 100000    COUNT STOPKEY
 100000     TABLE ACCESS FULL ACM203
```

```
SELECT SLP_STLCD
FROM
 MACCOUNT.ACM201  WHERE FINAC_YYMM = :b1  AND FINAC_DY = :b2  AND REGDP_CD =
  :b3  AND SLP_SRNO = :b4


call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute 100000      5.98       5.15          0          0          0           0
Fetch   100000      3.96       3.46          0     400000          0      100000
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total   200001      9.94       8.61          0     400000          0      100000
```

## Scalar SubQuery Expression의 성능

### Scalar SubQuery 방식 일때

```
select count(slp_stype)
from (select (select slp_stlcd from maccount.acm201 b
              where  b.finac_yymm = a.finac_yymm
              and    b.finac_dy   = a.finac_dy
              and    b.regdp_cd   = a.regdp_cd
              and    b.slp_srno   = a.slp_srno ) slp_stype,
             a.finac_yymm, a.finac_dy, a.regdp_cd, slp_srno
       from   maccount.acm203 a
      where rownum <=100000)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.66       0.65          0      62229         12           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.67       0.66          0      62229         12           1

Misses in library cache during parse: 1
Optimizer goal: CHOOSE
Parsing user id: 61  (ADMACCOUNT)

Rows     Row Source Operation
-------  ---------------------------
      1  SORT AGGREGATE
 100000   VIEW
 100000    COUNT STOPKEY
 100000     TABLE ACCESS FULL ACM203
```

| 구분 | CPU [sec] | QUERY |
|---|---|---|
| Join(NL) | 2.28 | 402,501 |
| Function | 22.75 | 2,501 |
| Scalar | 0.67 | 62,229 |

## Scalar SubQuery Expression의 사용가이드

### 적용범위

1) 메인 테이블과 여러 개의 코드를 공통코드와 조인하여 코드명을
   보여주고자 할 때
2) N:1관계에서 1쪽에 1개의 column만 조회될 때(마이그레이션)
3) 스타-스키마 테이블의 조인
4) 처리 속도가 조인에 의한 처리보다 빠르게 처리가 되므로 여러 가지 응용이 가능함

### 코드성 테이블에서 명칭만 가져오는 경우

```
SELECT a.regino,
       a.recevymd,
       a.recevno,
       a.regipocd,
      (SELECT ponm
       FROM   picmt0060 y
       WHERE y.regipocd = a.regipocd
       AND      y.useyn = 'Y') regipocdnm
FROM    PRRWT0110 a
WHERE   a.regino = '1696601000160';
```

## Scalar SubQuery Expression의 사용가이드

단순히 **count, sum**등의 간단한 **Aggregate** 작업인 경우

```
SELECT a.recevno,
       a.recevymd,
       a.regipocd,
       (SELECT count(1)
        FROM prrwt0110 y
        WHERE y.recevymd = a.recevymd
        AND   y.recevno = a.recevno) regicnt
FROM   PRRWT0010 a
WHERE  a.regipocd = '10024'
AND    a.recevymd = '20030701'
AND    a.domregiyn = 'Y';
```

아크 관계인 키 엔터티의 테이블에서 명칭만 가져오는 경우

```
SELECT 제안번호, 제목, 내용,
       DECODE(제안출처구분,'사원',
                  (SELECT 성명 FROM 사원 b WHERE b.사번 = a.제안출처),
                  (SELECT 조직명 FROM   현조직_V c WHERE c.조직코드 = a.제안출처)) 제안출처
FROM   제안 a
```

## Scalar SubQuery Expression의 사용가이드

코드성 테이블의 조인이 많아 실행계획이 복잡해져서 제어가 어려운 경우

```
SELECT a.recevymd,  b.recevhms,  a.regipocd,
              (SELECT ponm
               FROM picmt0060 y
               WHERE y.regipocd = a.regipocd
               AND   y.useyn = 'Y') regipocdnm,
              (SELECT y.comncdshortnm
               FROM picmt0040 y
               WHERE y.largedivcd = 'C20'
               AND   y.middivcd = '025'
               AND   y.comncd = a.daynightdivcd
               AND   y.useyn = 'Y') daynightdivcdnm,
              (SELECT y.comncdshortnm
               FROM picmt0040 y
               WHERE y.largedivcd = 'C20'
               AND   y.middivcd = '012'
               AND   y.comncd = a.prcpaymethcd
               AND   y.useyn = 'Y') prcpaymethcdnm,
              c.nm sendprsnnm
FROM PRRWT0100 a, prrwt0010 b, prrwt0000 c
WHERE a.regipocd = '11709'
AND   a.recevymd = '20030701'
AND   a.recevno = b.recevno
AND   a.sendprsnaddrseq = c.addrseq(+);
```

## Scalar SubQuery Expression의 사용가이드

복잡한 쿼리의 실행계획 단순화를 위해 사용하는 경우

```
select a.apprno,
       (SELECT /*+ INDEX_DESC(b PRCTT0050_PK) */
               (SELECT y.zipcd || ' ' || y.citydivnm || ' ' || y.sidivnm || ' ' || y.dlnm || ' '
                FROM    picmt0080 y
                WHERE  y.zipcd = b.sendprsnzipcd
                AND       y.useyn = 'Y') || b.sendprsnzipcdunderaddr
         FROM   PRCTT0050 b
         WHERE  b.apprno = a.apprno  AND   rownum =1   ) sendprsnaddr,
        (SELECT y.comncdshortnm
         FROM    picmt0040 y
         WHERE  y.largedivcd = 'CB0'            AND   y.middivcd = '012'
         AND       y.comncd = a.cntracdivcd  AND   y.useyn = 'Y') cntracdivnm,
        (SELECT y.comncdshortnm
         FROM    picmt0040 y
         WHERE  y.largedivcd = 'C60'    AND   y.middivcd = '004'
         AND       y.comncd = a.stuscd   AND   y.useyn = 'Y') prcpaymethcdnm
from   PRCTT0040 a
where  a.apprno = '1001510069'
```

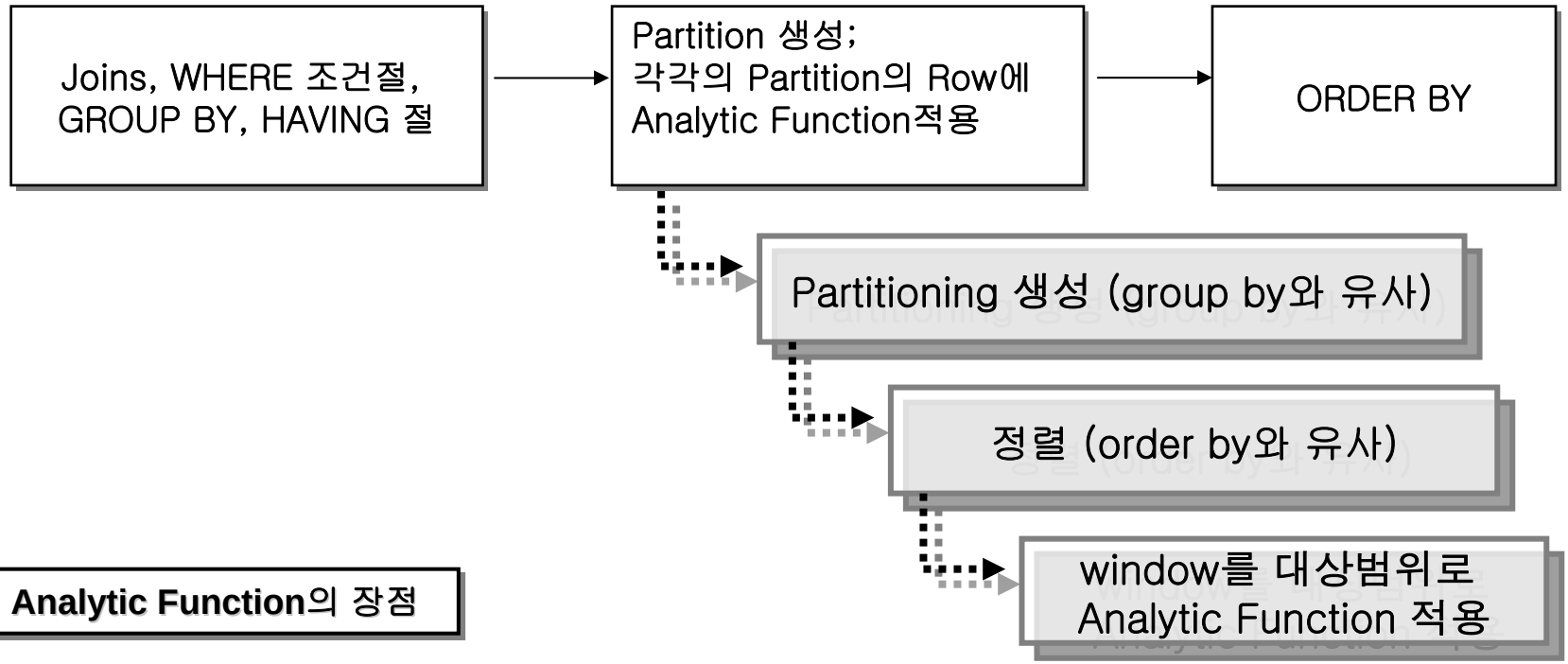
## Analytic Functions 이란

- **RUNNING SUMMARY, MOVING AVERAGE, RANKING, LEAD/LAG COMPARISION** 등 **BUSINESS** 분야에서 자주 행하여지는 여러 가지 형태의 분석에 유용하게 활용될 수 있는 **SQL function**
- 각 **window**별 집합 연산을 수행한 결과를 **return**하는 함수
- **ANSI** 표준을 따르고 있으며**, Oracle 8i**의 **New Feature**로서 제공**.**

## *SYNTAX*

***Analytic Function*** **(***arguments***)**
**OVER (PARTITION BY** *column_name* **[,** *column_name***]…**
**ORDER BY** *column_name* **[ASC|DESC][NULLS FIRST|LAST] [,** *column_name***]…**
*windowing_clause* **)**

**수행절차**




**Analytic Function의 장점**

- ❑ **Query speed**의 향상 : **Self-join,** 절차적 로직으로 표현한 것을 **native SQL**에서 바로 적용할 수 있도록 하여 **Join**이나 프로그램의 **Over Head**를 줄임
- ❑ 향상된 개발 생산력 : 간결한 **SQL**로 복잡한 분석작업을 수행 가능하며, 유지보수가 간편하여 생산성 향상
- ❑ 기존 **SQL syntax**를 그대로 따르기 때문에 이해 및 활용이 용이
- ❑ **ANSI SQL**로 채택될 것이므로 다양한 소프트웨어에 적용이 가능

## Analytic Function 의 종류

| 함수 | 설명 | 비고 |
|---|---|---|
| AVG | 평균값 구하기 | |
| COUNT | Query 결과의 개수 구하기 | |
| MIN | 최소값 구하기 | |
| SUM | 합 구하기 | |
| RANK | 각 row들의 ORDER BY 절에 의한 순위 구하기 | 같은 값에 대해서는 같은 순위를 매기며,<br>같은 순위 그 다음 순위가 생략됨 |
| DENSE_RANK | 각 row들의 순위 구하기 | 같은 값에 대해서는 같은 순위를 매기며,<br>같은 순위 그 다음 순위가 생략되지 않음 |
| CUME_DIST | 최저 값과 최고 값 사이에서의 상대적 위치 | 0 < CUME_DIST <= 1 |
| PERCENT_RANK | CUME_DIST와 비슷<br>(rank of row in its partition – 1) / (number of rows in the partition – 1) | 0 <= CUME_DIST <= 1 |

## Analytic Function 의 종류

| 함수 | 설명 | 비고 |
|---|---|---|
| LAG | 주어진 offset만큼 이전의 위치에 있는 데이터를 가져오기 | LAG(*column_name* [, *offset*] [, *default*])<br>default *offset* = 1<br>default *default* = null |
| LEAD | 주어진 offset만큼 다음의 위치에 있는 데이터를 가져오기 | LEAD<br>(*column_name* [, *offset*] [, *default*])<br>default *offset* = 1<br>default *default* = null |
| NTILE | 정렬된 데이터를 버켓 수만큼 나눠 각 row에 버켓 넘버를 지정 | |
| FIRST_VALUE | 정렬된 결과집합의 첫번째 값 | |
| LAST_VALUE | 정렬된 결과집합의 마지막 값 | |
| ROW_NUMBER | 파티션 또는 결과집합 내에서의 각 row에 유일한 번호를 지정 | |

## Analytic Function 예제

예제 : 그룹별 순위 매기기

```
SQL> SELECT dept_id, last_name, salary,
                ROW_NUMBER() OVER
                (PARTITION BY dept_id ORDER BY salary DESC, commission_pct) as rk
       FROM s_emp;
```

| DEPT_ID | LAST_NAME | SALARY | RK |
|---|---|---|---|
| 10 | Quick-To-See | 1450 | 1 |
| 31 | Magee | 1400 | 1 |
| 31 | Nagayama | 1400 | 2 |
| 32 | Giljum | 1490 | 1 |
| 33 | Sedeghi | 1515 | 1 |
| 34 | Nguyen | 1525 | 1 |
| 34 | Patel | 795 | 2 |
| 35 | Dumas | 1450 | 1 |
| …… | …… | …… | … |

## Analytic Function 예제

예제 : 그룹별 소계 구하기

```
SQL> SELECT manager_id, last_name, salary,
            SUM(salary) OVER (PARTITION BY manager_id ORDER BY salary
            RANGE UNBOUNDED PRECEDING) l_csum
     FROM s_emp;
```

| MANAGER_ID | LAST_NAME | SALARY | L_CSUM |
|---|---|---|---|
| 1 | Nagayama | 1400 | 1400 |
| 1 | Ngao | 1450 | 4300 |
| 1 | Quick-To-See | 1450 | 4300 |
| 1 | Ropeburn | 1550 | 5850 |
| 2 | Biri | 1100 | 1100 |
| 2 | Urguhart | 1200 | 2300 |
| 2 | Menchu | 1250 | 3550 |
| 2 | Catchpole | 1300 | 4850 |
| 2 | Havel | 1307 | 6157 |
| 3 | Magee | 1400 | 1400 |
| 3 | Dumas | 1450 | 2850 |
| 3 | Giljum | 1490 | 4340 |
| 3 | Sedeghi | 1515 | 5855 |
| 3 | Nguyen | 1525 | 7380 |
| 6 | Smith | 940 | 940 |
| 6 | Maduro | 1400 | 2340 |
| …… | …… | …… | …… |

## Analytic Function 예제

예제 : 이전 값, 다음 값 구하기

```
SQL>  SELECT last_name, start_date,
                  LAG(start_date, 1) OVER (ORDER BY start_date) AS last_hire_date,
                  LEAD(start_date, 1) OVER (ORDER BY start_date) AS next_hire_date
        FROM s_emp;
```

| LAST_NAME | START_DATE | LAST_HIRE_DATE | NEXT_HIRE_DATE |
|---|---|---|---|
| Ropeburn | 2090-03-04 | | 2090-03-08 |
| Smith | 2090-03-08 | 2090-03-04 | 2090-04-07 |
| Quick-To-See | 2090-04-07 | 2090-03-08 | 2090-04-07 |
| Biri | 2090-04-07 | 2090-04-07 | 2090-05-03 |
| Velasquez | 2090-05-03 | 2090-04-07 | 2090-05-08 |
| Ngao | 2090-05-08 | 2090-05-03 | 2090-05-14 |
| Menchu | 2090-05-14 | 2090-05-08 | 2090-05-14 |
| Magee | 2090-05-14 | 2090-05-14 | 2090-10-17 |
| Patel | 2090-10-17 | 2090-05-14 | 2090-11-30 |
| ..... | ..... | ..... | ..... |

## Analytic Function

Window size정의에 **사용되는** 키워드 사례

```
SELECT deptno,job,empno, ename, hiredate, sal,
    SUM(sal) over (PARTITION BY deptno ORDER BY job) 누계
FROM emp
```

**SUM(sal)** 함수의 처리 대상 범위

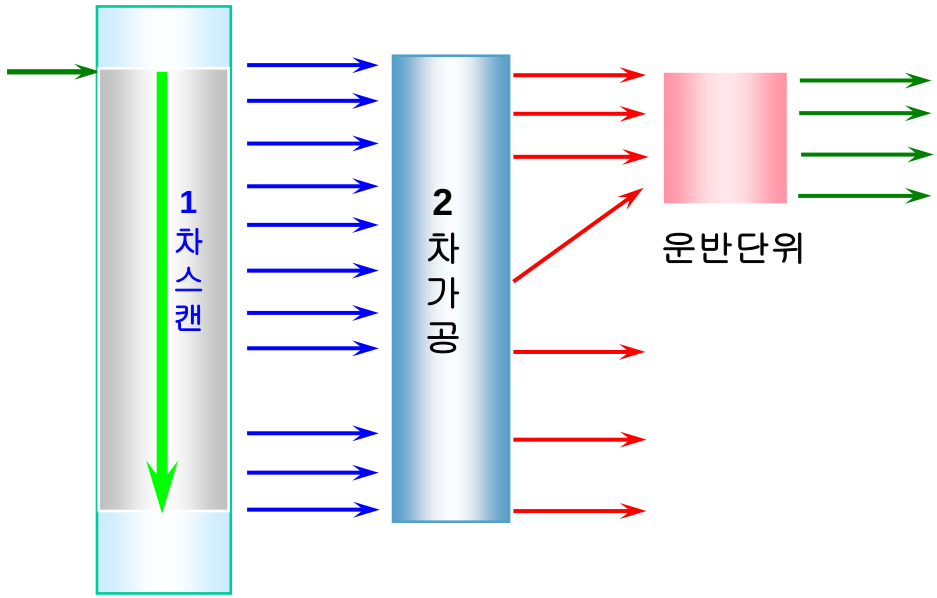
**WINDOW**

| DEPTNO | JOB | EMPNO | ENAME | HIREDATE | SAL | 누계 |
|---|---|---|---|---|---|---|
| 10 | CLERK | 7934 | MILLER | 1982-01-23 | 1,300 | 1,300 |
| 10 | MANAGER | 7782 | CLARK | 1981-06-09 | 2,450 | 3,750 |
| 10 | PRESIDENT | 7839 | KING | 1981-11-17 | 5,000 | 8,750 |
| 20 | ANALYST | 7788 | SCOTT | 1982-12-09 | 3,000 | 3,000 |
| 20 | ANALYST | 7902 | FORD | 1981-12-03 | 3,000 | 6,000 |
| 20 | CLERK | 7369 | SMITH | 1980-12-17 | 800 | 6,800 |
| 20 | CLERK | 7876 | ADAMS | 1983-01-12 | 1,100 | 7,900 |
| 20 | MANAGER | 7566 | JONES | 1981-04-02 | 2,975 | 10,875 |
| 30 | CLERK | 7900 | JAMES | 1981-12-03 | 950 | 950 |
| 30 | MANAGER | 7698 | BLAKE | 1981-05-01 | 2,850 | 3,800 |
| 30 | SALESMAN | 7499 | ALLEN | 1981-02-20 | 1,600 | 5,400 |
| 30 | SALESMAN | 7654 | MARTIN | 1981-09-28 | 1,250 | 6,650 |
| 30 | SALESMAN | 7844 | TURNER | 1981-09-08 | 1,500 | 8,150 |
| 30 | SALESMAN | 7521 | WARD | 1981-02-22 | 1,250 | 9,400 |

**Partition**

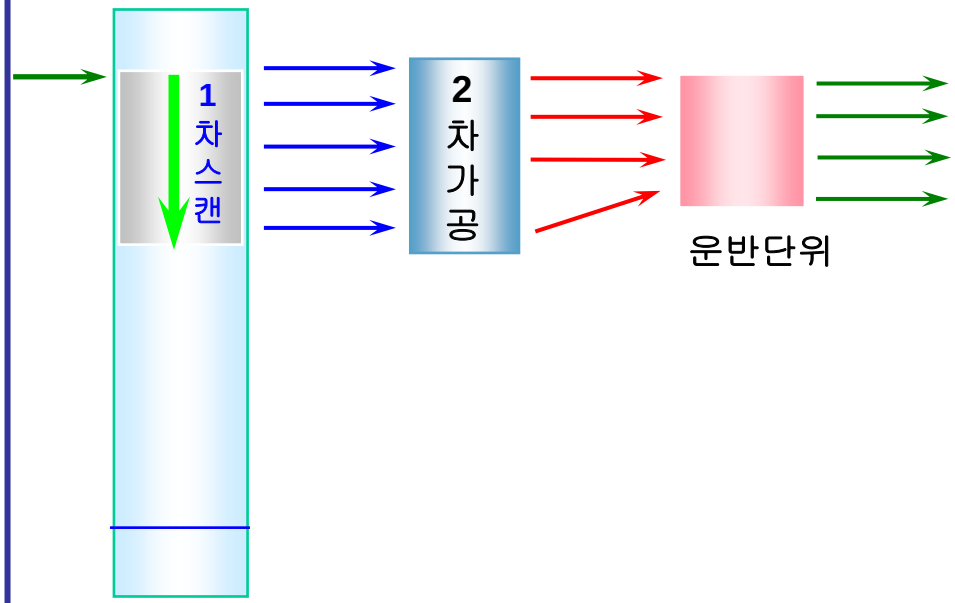
## 부분범위처리의 정의

### 전 체 범 위 처 리




**Full Range Scan** 후 가공하여 **Array Size** 만큼 추출

### 부 분 범 위 처 리




조건을 만족하는 **Row** 수가 **Array Size** 에 도달되면 멈춤

## 부분범위처리의 정의

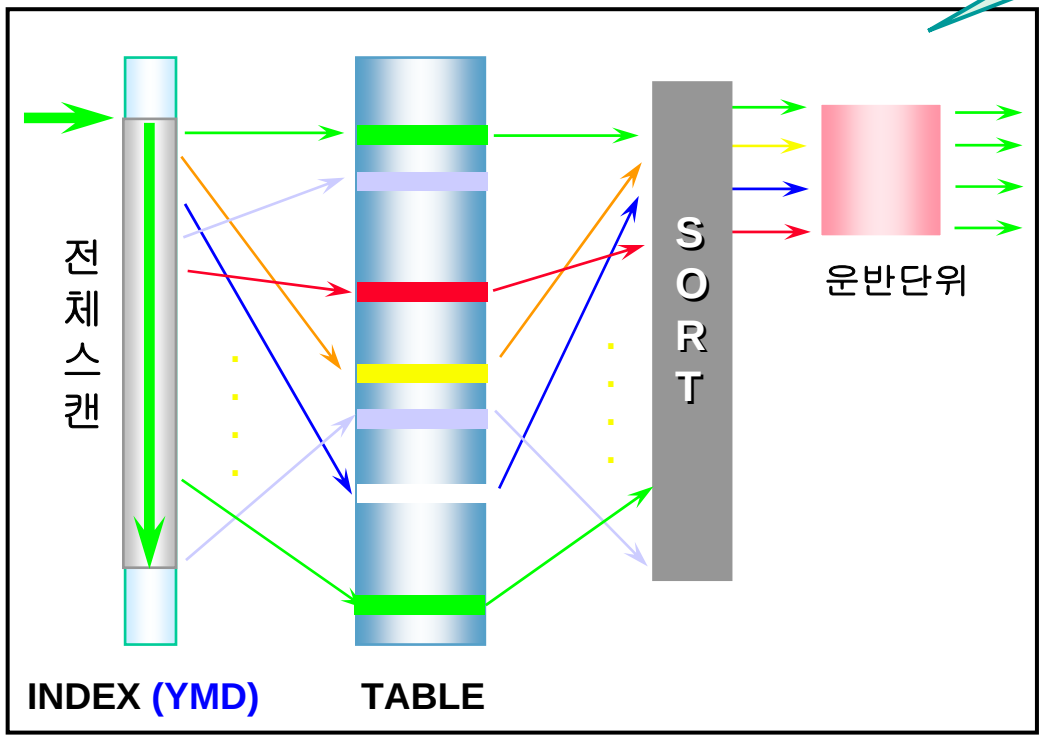
### 개념 및 적용 원칙

- ❑ 조건을 만족하는 전체집합이 아닌 일부분만 ACCESS
- ❑ DATA양이 많아도 PERFORMANCE에 지장이 없고, 오히려 향상
- ❑ INDEX나 CLUSTER를 적절히 활용한 SORT의 대체
- ❑ MAX 처리
- ❑ TABLE은 ACCESS 하지 않고 INDEX만 사용하도록 유도
- ❑ EXISTS의 활용
- ❑ ROWNUM의 활용
- ❑ Stored Function을 이용
- ❑ Scalar SubQuery을 이용

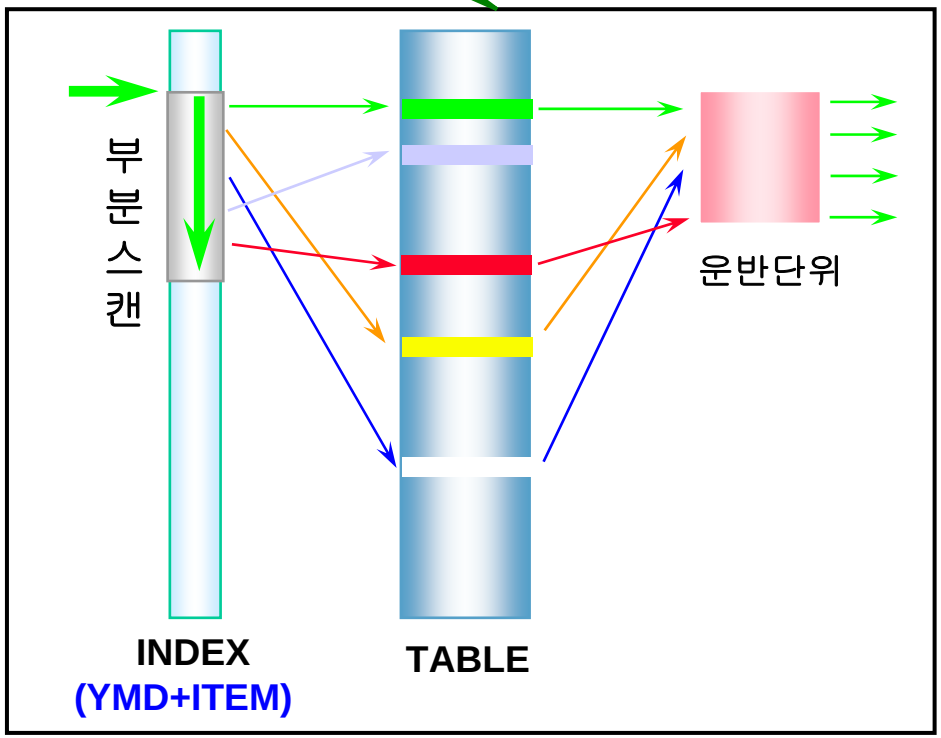
## 부분범위처리 유형

**SORT를 대신하는 INDEX**

```
SELECT *
  FROM  PRODUCT
 WHERE  YMD = '951023'
   AND  ITEM LIKE 'AB%'
ORDER BY YMD, ITEM
```




```
SELECT *
  FROM  PRODUCT
 WHERE  YMD = '951023'
   AND  ITEM LIKE 'AB%';
```



## 부분범위처리 유형

### SORT 대체 사례

```
SQL> SELECT ORDDATE, CUSTNO
       FROM ORDER1T
      WHERE ORDDATE between '940101' and '941130'
      ORDER BY ORDDATE  DESC
```

```
21200   SORT ORDER BY
21201     INDEX RANGE SCAN ORDDATE

ORDDATE index : ORDDATE + CUSTNO
```

5.2 sec

```
SQL> SELECT /*+ INDEX_DESC(A ORDDATE)  */
         ORDDATE, CUSTNO
       FROM ORDER1T A
      WHERE ORDDATE between '940101' and '941130'
```

```
20      INDEX RANGE SCAN DESCENDING ORDDATE

ORDDATE index : ORDDATE + CUSTNO
```

0.01 sec

인덱스 구조 변경 및 조건 추가

```
SQL> SELECT ORDDATE, CUSTNO
       FROM ORDER1T
      WHERE ORDDEPT LIKE '7%'
      ORDER BY ORDDATE  DESC
```

```
42000  SORT ORDER BY
42001    INDEX RANGE SCAN  ORDDEPT

ORDDEPT index : ORDDEPT + ORDDATE + CUSTNO
```

12.5 sec

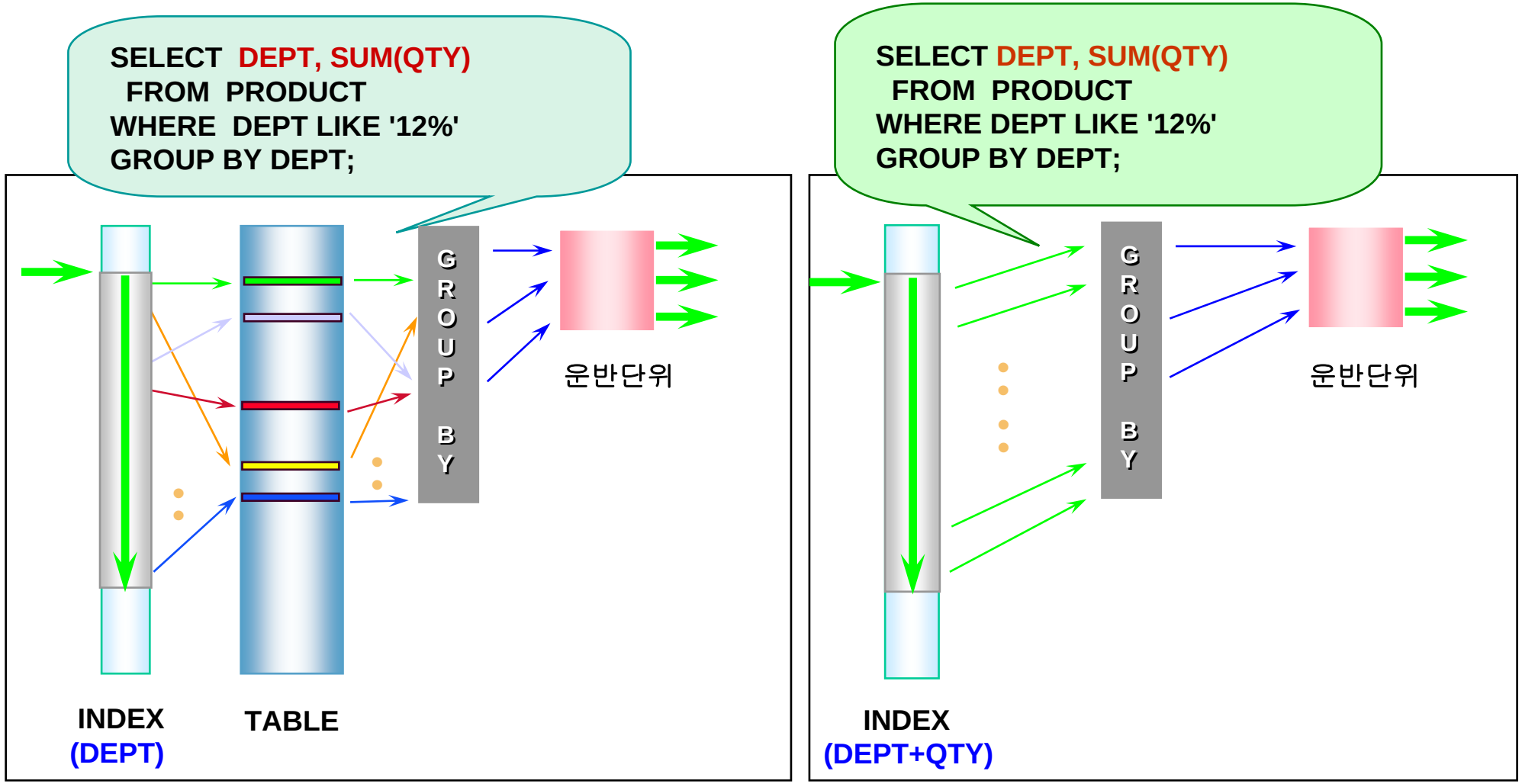
```
SQL> SELECT /*+ INDEX_DESC(A ORDDATE)  */
         ORDDATE, CUSTNO
       FROM ORDER1T A
      WHERE ORDDEPT LIKE '7%'
        AND ORDDATE <= '991231'
```

```
20      INDEX RANGE SCAN DESCENDING ORDDATE

ORDDATE index : ORDDATE + ORDDEPT + CUSTNO
```

0.02 sec

## 부분범위처리 유형

**INDEX만 처리**

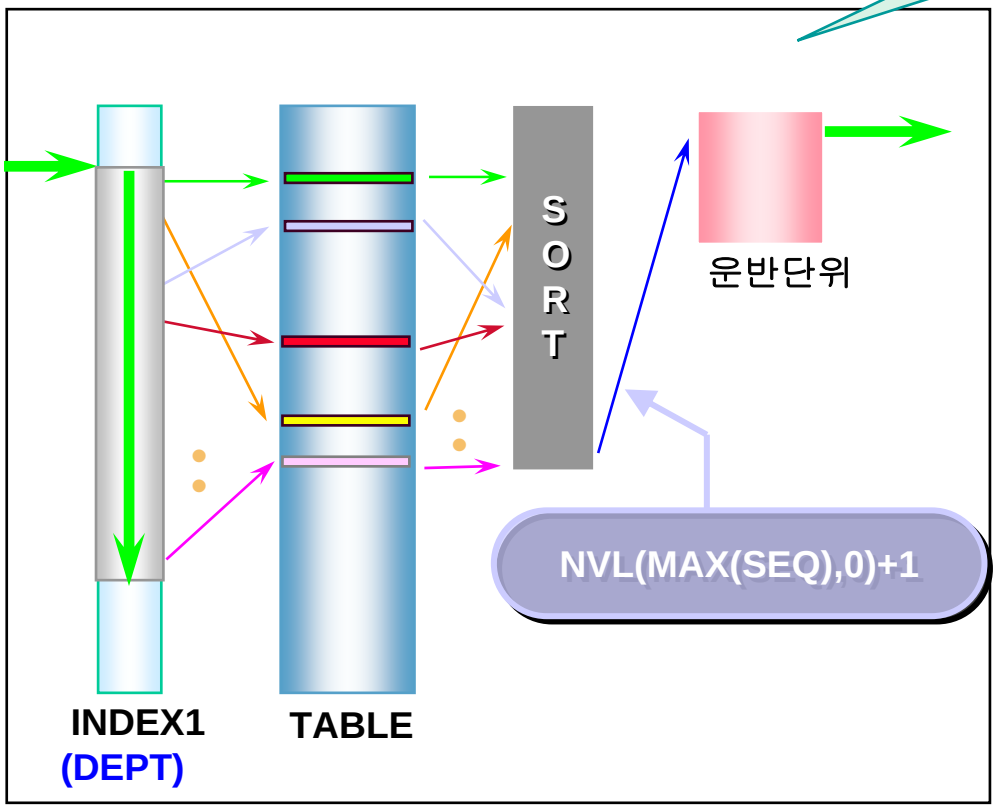


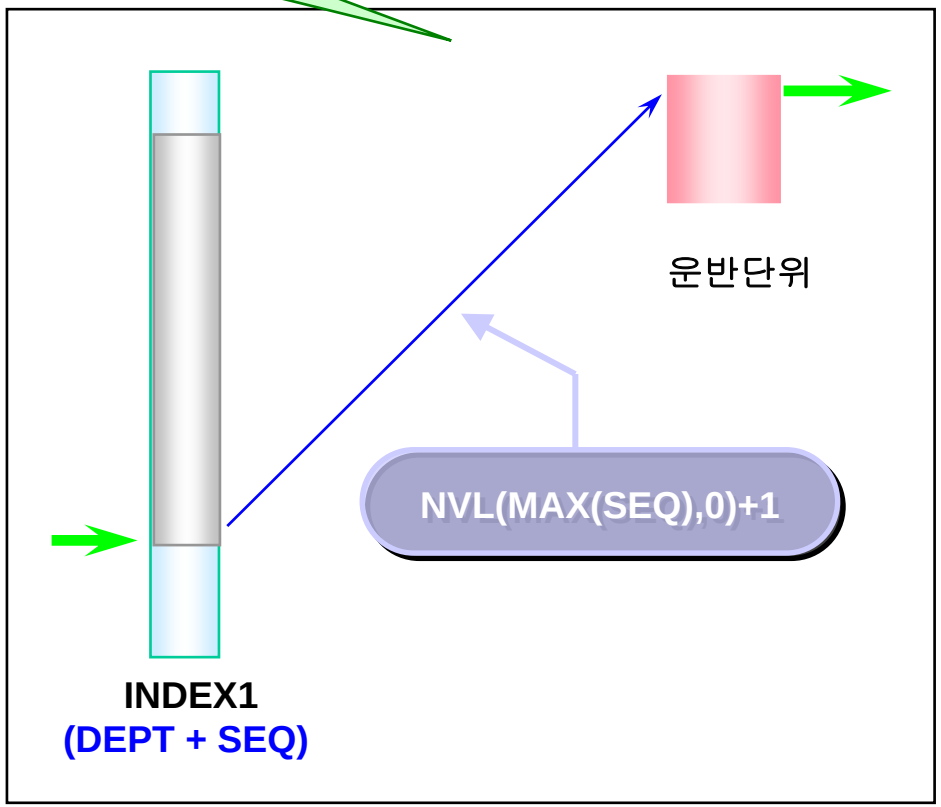
## 부분범위처리 유형

### MAX 처리

```
SELECT  NVL(MAX(SEQ), 0) + 1
  FROM  PRODUCT
WHERE  DEPT = '12300';
```

```
SELECT /*+ INDEX_DESC( A INDEX1) */
       NVL(MAX(SEQ),0) + 1
  FROM  PRODUCT  A
WHERE  DEPT = '12300'
   AND  ROWNUM = 1;
```



## 부분범위처리 유형

**SQL 예제**

인덱스만 처리

```
SQL>  SELECT STATUS, COUNT(*)
        FROM ORDER2T
       WHERE ITEM LIKE 'HJ%'
       GROUP BY STATUS
```

10.3 sec

```
  20  SORT GROUP BY
36630    TABLE ACCESS BY  INDEX  ROWID ORDER2T
36631      INDEX RANGE SCAN ITEM_IDX1

 ITEM_IDX1 index : ITEM
```

```
SQL>  SELECT STATUS, COUNT(*)
        FROM ORDER2T
       WHERE ITEM LIKE 'HJ%'
       GROUP BY STATUS
```

2.5 sec

```
  20  SORT GROUP BY
36631    INDEX RANGE SCAN ITEM_IDX1

 ITEM_IDX1 index : ITEM + STATUS
```

MAX 처리

```
SQL>  SELECT NVL(MAX(ORDDATE),'없음
        FROM ORDER1T
       WHERE ORDDEPT = '430'
         AND STATUS = '30'
```

2.53 sec

```
   1  SORT AGGREGATE
2892    TABLE ACCESS BY INDEX ROWID ORDER1T
15230     INDEX RANGE SCAN DEPT_DATE

DEPT_DATE index : ORDDEPT + ORDDATE
```

```
SQL> SELECT /*+ INDEX_DESC(A dept_date)  */
            NVL(MAX(ORDDATE),'없음')
       FROM ORDER1T A
      WHERE ORDDEPT = '430' AND STATUS = '30'
        AND ROWNUM = 1
```

0.01 sec

```
1   COUNT STOPKEY
1     TABLE ACCESS BY INDEX ROWID ORDER1T
1       INDEX RANGE SCAN DESCENDING DEPT_DATE

DEPT_DATE index : ORDDEPT + ORDDATE
```

## 부분범위처리 유형

**MAX 처리시 RBO와 CBO의 차이(Oracle 8i 이상)**

### RBO인 경우(Oracle 8i 이상)

```
SQL> SELECT  MAX(JOIN_DT)
      FROM  MEMBERT
     WHERE  ADR_ZIP ='10278'
```

10.2 sec

```
   1  SORT AGGREGATE
20001    INDEX RANGE SCAN  MEMBERT_IDX05

MEMBERT_IDX05  index :  ADR_ZIP + JOIN_DT
```

### CBO인 경우(Oracle 8i 이상)

```
SQL> SELECT  MAX(JOIN_DT)
      FROM  MEMBERT
     WHERE  ADR_ZIP ='10278'
```

0.01 sec

```
1  SORT AGGREGATE
1    FIRST ROW
1      INDEX RANGE  SCAN(MIN/MAX) MEMBERT_IDX05

MEMBERT_IDX05  index :  ADR_ZIP + JOIN_DT
```

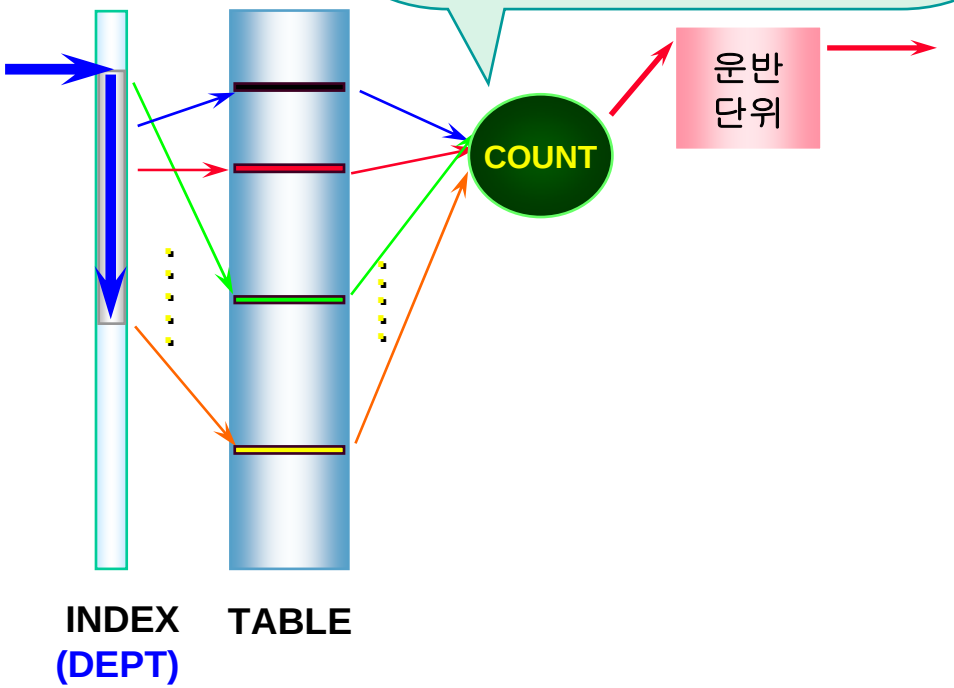
### MIN/MAX AGGREGATE OPTIMIZATION

- ❑ 인덱스 컬럼을 이용한 질의여야 함
- ❑ 한 개의 MIN 또는 MAX 값만 포함되어야 함
- ❑ 한 개의 'RANGE'만을 이용한 질의여야 함
- ❑ Sorted Index에서만 가능하며, 인덱스 키의 정렬 순서와는 무관

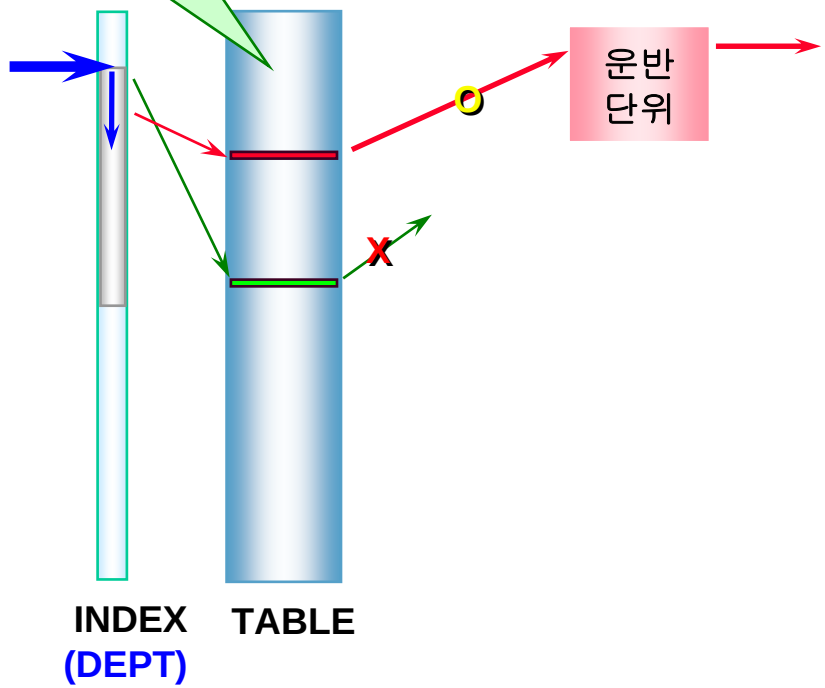
## 부분범위처리 유형

**EXISTS 활용**

```
SELECT COUNT(*)  INTO :CNT
  FROM  ITEM_TAB
WHERE  DEPT = '101'
   AND  SEQ > 100

IF CNT > 0 . . .
. . . . . .
```



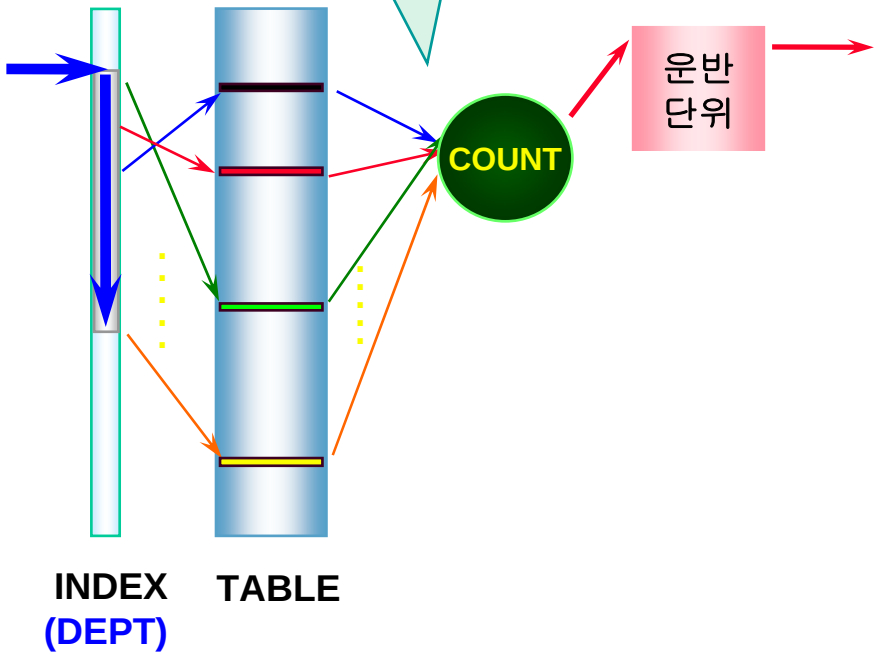

```
SELECT 1 INTO :CNT  FROM  DUAL
WHERE EXISTS
      ( SELECT  'X'
        FROM  ITEM_TAB
       WHERE  DEPT = '101'
         AND  SEQ  >  100 )

IF CNT > 0 . . .
```



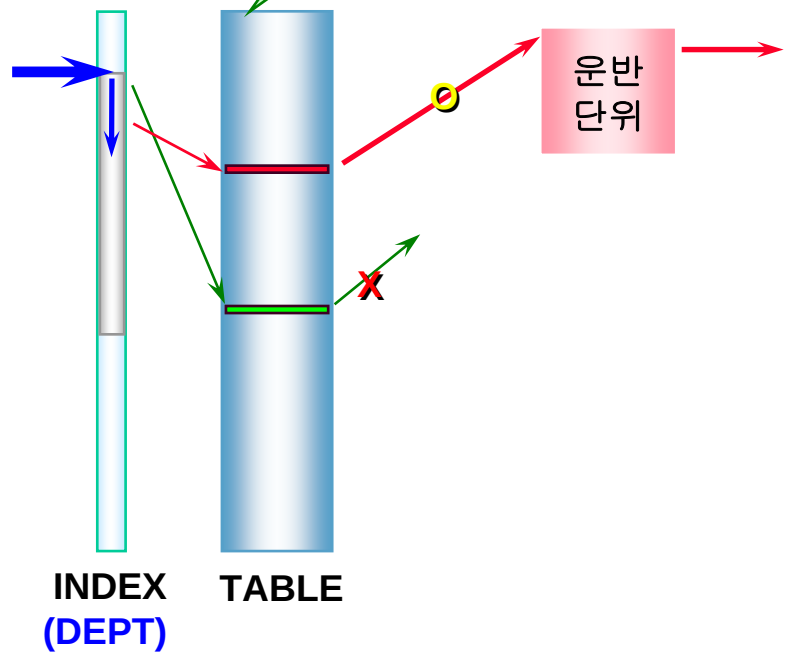
## 부분범위처리 유형

**ROWNUM 활용**

```
SELECT COUNT(*)  INTO :CNT
  FROM  ITEM_TAB
WHERE  DEPT = '101'
    AND  SEQ > 100
. . . . . .
IF CNT > 0 . . .
. . . . . .
```




```
SELECT 1 INTO :CNT
  FROM   ITEM_TAB
 WHERE  DEPT = '101'
    AND  SEQ  > 100
    AND  ROWNUM = 1
. . . . . .
IF CNT > 0
. . . . . .
```



## 부분범위처리 유형

**1:M 조인의 부분범위처리 유도 (Sub-Query로 처리)**

```
SELECT x.CUST_NO, x.ADDR, x.NAME, .............
  FROM CUST x,  REQT  y
 WHERE x.CUST_NO = y.CUST_NO
   AND x.CUST_STAT in ('A', 'C', 'F')
   AND y.UN_PAY > 0
 GROUP BY  x.CUST_NO
HAVING SUM(y.UN_PAY)  between  :VAL1 and :VAL2
```

**CUST_NO는 CUST 테이블의 PK INDEX임**

전체범위

부분범위

```
SELECT x.CUST_NO, x.ADDR, x.NAME, .............
  FROM CUST x
 WHERE CUST_STAT in ('A', 'C', 'F')
   AND EXISTS ( SELECT X'
                  FROM REQT y
                 WHERE y.CUST_NO = x.CUST_NO
                   AND y.UN_PAY > 0
                 GROUP BY  x.CUST_NO
                HAVING SUM(y.UN_PAY) between  :VAL1 and :VAL2 )
```

**SUB_QUERY** 의 수행결과를 **MAIN_QUERY** 에서 사용할 수 없음

## 부분범위처리 유형

### 1:M 조인의 부분범위처리 유도 (함수로 처리)

```
SELECT x.CUST_NO, x.ADDR, x.NAME, .............
  FROM CUST x,  REQT  y
 WHERE x.CUST_NO = y.CUST_NO
   AND x.CUST_STAT in ('A', 'C', 'F')
   AND y.UN_PAY > 0
 GROUP BY  x.CUST_NO
HAVING SUM(y.UN_PAY)  between  :VAL1 and :VAL2
```

전체범위

부분범위

```
Create or replace Function unpay_sum
           (v_custno  in varchar2)
    return number is
    sum_unpay number ;
begin
    ..............
    select sum(un_pay) into sum_unpay
      from reqt
     where cust_no = v_custno
       and un_pay > 0 ;
    ..............
    return sum_unpay ;
end unpay_sum ;
```

```
SELECT CUST_NO, ADDR, UN_PAY, .....
FROM ( SELECT CUST_NO, ADDR,
              UNPAY_SUM(CUST_NO) AS UN_PAY,
                        ....
         FROM CUST
        WHERE CUST_STAT IN ('A', 'C', 'F') )
WHERE UN_PAY BETWEEN  :VAL1 AND :VAL2
```

## 부분범위처리 유형

**1:M 조인의 부분범위처리 유도 (Scalar Subquery 처리)**

```
SELECT x.CUST_NO, x.ADDR, x.NAME, .............
  FROM CUST x,  REQT  y
 WHERE x.CUST_NO = y.CUST_NO
   AND x.CUST_STAT in ('A', 'C', 'F')
   AND y.UN_PAY > 0
 GROUP BY  x.CUST_NO
HAVING SUM(y.UN_PAY)  between  :VAL1 and :VAL2
```

전체범위

부분범위

```
SELECT cust_no, addr, un_pay, ...........
  FROM (SELECT x.cust_no, x.addr,
               (SELECT SUM(y.un_pay)
                  FROM reqt y
                 WHERE x.cust_no = y.cust_no
                   AND y.un_pay > 0)  AS un_pay,....
          FROM cust x
         WHERE x.cust_stat IN ('A', 'C', 'F') )
 WHERE un_pay BETWEEN  :VAL1 AND :VAL2
```

## DYNAMIC SQL 사용 시의 트레이스 형태

```
SELECT A.SCRBR_NO, A.APLN_NM, A.APLN_TEL_NO, A.AS_APLN_CL_CD, A.AS_WK_STAT_CD,
       A.RQST_DH, A.INSTAL_배송지코드, A.RECV_배송지코드, A.AS_RECV_NO,
       B.RECV_MTH_CD, B.SCRBR_NM, B.INDC_배송지코드,
       C.AS_ORDER_NO, C.WK_PRDIT_DT, D.SC_ID, F.SVC_OPEN_USER_ID
  FROM CMDA01T01 A, CMCC01T01 B, CMDB04T01 C, CMCC04T01 D, CMBB01T01 E, CMCB01T01 F
 WHERE A.SCRBR_NO    = B.SCRBR_NO
   AND A.AS_RECV_NO  = C.AS_RECV_NO
   AND A.SCRBR_NO    = D.SCRBR_NO
   AND A.SCRBR_NO    = E.SCRBR_NO
   AND E.CHG_KIND_CD = '101'
   AND E.RECV_NO     = F.RECV_NO
   AND D.SEQ_NO      = 1
   AND A.INSTAL_배송지코드 = 'L10277'
   AND A.AS_WK_STAT_CD     = '01'
   AND A.INSTAL_배송지코드 = 'L10277'
ORDER BY C.AS_RECV_NO DESC
```

심각한 **PARSING** 부하 !!

| call | count | cpu | elapsed | disk | query | current | rows |
|---|---|---|---|---|---|---|---|
| Parse | 20 | 0.81 | 1848.30 | 0 | 0 | 0 | 0 |
| Execute | 20 | 0.01 | 0.01 | 0 | 0 | 0 | 0 |
| Fetch | 20 | 0.74 | 1.58 | 114 | 15693 | 0 | 50 |
| total | 60 | 1.56 | 1849.89 | 114 | 15693 | 0 | 50 |

## 실행계획 분리

```
SELECT SUBSTR(매출일,1,6), SUM(매출액)
       SUM(손익액)
FROM   매출손익
WHERE  ( :IN_CUST <> '101'                  and
         거래처코드 = :IN_CUST              and
         매출일 LIKE :IN_DATE||'%' )
   OR  ( :IN_CUST = '101'                   and
         RTRIM(거래처코드) = :IN_CUST and
         매출일 like :IN_DATE||'&' )
```

인덱스 스캔 OR FULL 스캔
= FULL 스캔

인덱스 스캔

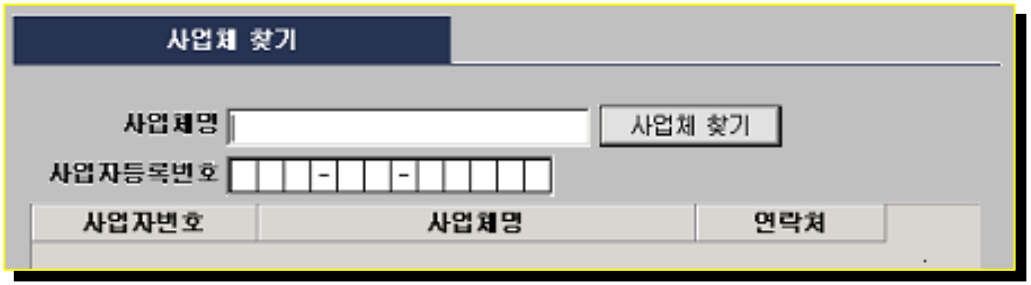
```
SELECT SUBSTR(매출일,1,6), SUM(매출액)
       SUM(손익액)
FROM   매출손익
WHERE  :IN_CUST <> '101'
   and  거래처코드 = :IN_CUST
   and  매출일 LIKE :IN_DATE||'%' )
UNION ALL
SELECT SUBSTR(매출일,1,6), SUM(매출액)
       SUM(손익액)
FROM   매출손익
WHERE  :IN_CUST = '101'
   and  RTRIM(거래처코드) = :IN_CUST
   and  매출일 like :IN_DATE||'&' )
```

둘중 하나만 수행

FULL 스캔

## 실행계획 분리 사례

팝업 창을 열어 사업체를 조회하는 화면으로, 사업체명은 일부라도 반드시 입력해야 하고, 사업자등록번호는 필요에 따라 입력할 수 있다.




BUSI_UK_BNO : BUSI_NO + BUSI_ID
BUSI_UK_NM : BUSI_NM + BUSI_ID

```
SELECT BUSI_ID,
            SUBSTR(BUSI_NO,1,3)||'-'||
            SUBSTR(BUSI_NO,4,2)||'-'||
            SUBSTR(BUSI_NO,6,5),
            BUSI_NM, PHONE_NO1
  FROM BUSI
 WHERE BUSI_NO LIKE :BUSI_NO || '%'
   AND BUSI_NM LIKE :BUSI_NM || '%' ;
```

엑세스 량이 많다

```
0  SELECT STATEMENT   GOAL: CHOOSE
12643   TABLE ACCESS (BY INDEX ROWID) OF 'BUSI'
12644   INDEX (RANGE SCAN) OF 'BUSI_UK_BNO' (UNIQUE)
```

## 실행계획 분리 사례

```
SELECT BUSI_ID,
       ...
  FROM BUSI
 WHERE :SW = '1'
       AND BUSI_NO = :BUSI_NO
       AND BUSI_NM LIKE :BUSI_NM || '%'
 UNION ALL
SELECT BUSI_ID,
       ...
  FROM BUSI
 WHERE :SW='2'
       AND BUSI_NM LIKE :BUSI_NM || '%' ;
```

사업자번호가 입력되었을 때

```
0  SELECT STATEMENT   GOAL: CHOOSE
1   UNION-ALL
1    FILTER
1     TABLE ACCESS (BY INDEX ROWID) OF 'BUSI'
1      INDEX (UNIQUE SCAN) OF 'BUSI_UK_BNO' (UNIQUE)
0    FILTER
0     TABLE ACCESS (BY INDEX ROWID) OF 'BUSI'
0      INDEX (RANGE SCAN) OF 'BUSI_UK_NM' (UNIQUE)
```

사업자번호가 입력되지 않았을 때

```
   0  SELECT STATEMENT   GOAL: CHOOSE
3357   UNION-ALL
   0    FILTER
   0     TABLE ACCESS (BY INDEX ROWID) OF 'BUSI'
   0      INDEX (UNIQUE SCAN) OF 'BUSI_UK_BNO' (UNIQUE)
3357    FILTER
3357     TABLE ACCESS (BY INDEX ROWID) OF 'BUSI'
3358      INDEX (RANGE SCAN) OF 'BUSI_UK_NM' (UNIQUE)
```

# 6. 페이지 처리 SQL

## Web 페이지 처리 방식

❑ Web 또는 C/S 화면에서 사용자의 요구조건에 대한 결과 Set을 페이지(약 10 ~ 100건) 로 나누어, 해당 페이지별 목록을 보여주고 이동 가능하게 표현 하는 표시 방법을 페이지 처리 라 한다.

페이지 처리는 Client 지원/ SQL 지원의 2가지 방식이 있다.

❑ **Client 지원.**

2 Tier 로 연결된 C/S 프로그램들(예: Oracle Forms 등)에서 결과 SET에 대한 커서 이동 등을 통해 페이지 처리를 지원하는 기능으로 비효율이 없으나 현재의 Web 환경이나 대규모 접속 환경에서는 사용하지 않는(못하는) 방식.

❑ **SQL 페이지 처리.**

사용자가 SQL을 사용하여 매 페이지 마다 해당 페이지의 대상 결과을 가져와 보여주는 방식.

❑ **SQL 페이지 처리 구성.**

페이지 처리는 아래의 2가지 쿼리로 구성된다.

→ 카운트 쿼리(전체 대상 건수를 구해, 페이지 수를 구하기 위해 사용)

→ 페이지(리스트) 쿼리(해당 페이지에 보여줄 목록을 가져오는데 사용)

❑ **사용자 SQL 과 페이지 SQL**

사용자가 업무를 위해 클라이언트 툴에서 작성한 SQL(원 SQL) 과 페이지 처리를 위해 WAS에서 수행되는 SQL(카운트SQL 및 페이지 SQL)은 완전히 다른 SQL 이며, 각기 다르게 파싱 되며, 다른 실행계획이 수립되며, 다른 성능으로 수행된다. .

➔ **페이지 처리 화면은 페이지 처리에 맞는 페이지 SQL을 최적화 해야 성능을 보장 할 수 있다.**

## 페이지 처리 SQL 유형

❑ **최적화 SQL 방식**

전문컨설팅업체에서 제안하는 방식으로 화면 블럭(5 ~ 10개 페이지 단위) 단위로 엑세스 하여

각 페이지별 시작점의 조회값을 유지, 해당 페이지만을 가져오거나, 모든 SQL 을 최초 페이지, 다음 페이지, 이전 페이지, 최종 페이지 의 4개 SQL로 구성하는 방식으로 구성.

→ 장점 : 최적의 엑세스 가능.

→ 단점 : 엑세스 유형별 인덱스 구성 필요,

화면별 4개 SQL 작성 등에 따른 개발 시간 증가에 따른 생산성 하락 및 고 난이도 요구.

➔ 소수의 프로그램으로 다수의 화면(게시판 등)을 처리하는 경우 등에 사용시 유리.

❑ **ROWNUM STOP KEY 방식(누적 엑세스 방식)**

RDBMS에서 SQL의 결과 SET을 제한 하는 기능을 이용하는 방법.

오라클환경 에서는 ROWNUM(ROW_NUMBER( ) ) 을 이용하여 페이지 처리 SQL을 사용한다.

→ 장점 : 개발의 단순화, WEB 환경의 처리와 유사.

대부분의 개발 프로젝트 및 프레임웍에서 사용.

→ 단점 : 페이지별 누적 처리에 따라 페이지 증가시 처리량 증가.

(대부분의 웹화면 엑세스는 최초 몇 페이지 엑세스에 집중되므로 누적 부하는 무시)

❑ 차인뱅에서 사용할 페이지 처리 방식.

➔ **ROWNUM STOP KEY를 이용한 페이지 처리 방식 사용.**

## 페이지 처리 SQL 구성

❑ **카운트 SQL**

조회 화면의 첫 페이지(1 Page)에서 수행되며, 조회 조건에 대한 결과 건수를 구하는데 사용한다.

결과 건수는 '결과 건수 화면표시' 와 '총 페이지수' 를 구하는데 사용된다.

주의) 전체 건수를 읽어야 하므로 가능한 인덱스 만으로 처리되게 구성해야 한다.

→ 별도의 카운트 SQL 없이 원 SQL 외부에 인라인뷰로 처리하는 FrameWork도 다수 존재.

❑ **페이지 SQL**

첫 페이지에서부터 해당 페이지 마지막 리스트까지 누적 처리하여 마지막 페이지만을 리턴하여 화면에 표시한다.

❑ **페이지 SQL 엑세스 유형.**

→ 정렬 없이 엑세스 조건만 부여된 경우.

→ 정렬 조건과 사용자 조건이 동일하여 인덱스 정렬 특성을 이용한 엑세스로 대체한 경우.

→ 정렬 조건과 사용자 조건이 상이하여 전체 엑세스 후 정렬, 페이지 순서를 결정한 경우.

→ 집합 연산(UNION , MINUS 등)에 의해 전체 엑세스 후 (정렬), 페이지 순서를 결정한 경우.

→ 집계 연산(GROUP BY, DISTINCT 등)에 의해 전체 엑세스 후 (정렬), 페이지 순서를 결정한 경우.

→ SORT MERGE JOIN, HASH JOIN 등에 의해 전체, 혹은 일부 집합을 전체 처리한 경우.

→ Analytic Function 을 사용, 전체 결과를 대상으로 한 경우.

→ Scalar SubQuery 의 결과를 체크로 사용한 경우.

## 전체 범위 처리 되는 Case

- 페이지 처리란 말 그대로 **해당 페이지(까지)만** 처리한다는 의미이나 실제로는 전체 범위를 처리하는 경우.
  - ➔ 전체 범위 처리 원인은 작성된 SQL이 대상 전체를 모두 처리해야만 하는 **논리**를 가지고 있기 때문이다.
- 해당 페이지만을 처리하지 못하고 전체 범위 처리로 수행되는 이유는 대부분 ROWNUM의 특성에 있다.
  - ➔ **사용자 조건과 다른 ORDER BY 절 사용.**
  - ➔ **GROUP BY 절 , AGGREGATE 함수 사용.**
  - ➔ **SET 오퍼레이션(UNION, MINUS, INTERSECT 절) 사용.**
  - ➔ **ROWNUM 을 STOP KEY**가 아닌 **COUNT / FILTER** 로 사용.
  - ➔ **페이지 처리를 위한 핵심 집합 이외의 참조성 처리를 전체범위에서 수행하는 경우.**
- 페이지 처리 **구동 여부는** 실행계획과 SQL Trace 에서 **결과 Rows 를 확인**, 검증 할 수 있다.
- 페이지 처리 개념 자체에 **비효율이 존재** 하며, 이 **비효율을 최소화** 하는 것이 성능 향상의 **목표**이다.

```
Rows    Row Source Operation
     0  STATEMENT
   100   VIEW (cr=23181 pr=0 pw=0 time=124361 us)
   101    SORT ORDER BY (cr=23181 pr=0 pw=0 time=124459 us)
   101     WINDOW SORT PUSHED RANK (cr=23181 pr=0 pw=0 time=124363 us)
 23166      TABLE ACCESS BY INDEX ROWID ANAF_TEST (cr=23181 pr=0 pw=0 time=69510 us)
 23166       INDEX RANGE SCAN ANAF_TEST_X01 (cr=76 pr=0 pw=0 time=9 us)(Object ID 81665)
```

## 전체 범위 처리 되는 사례1

❑ 사용자 조건과 다른 **ORDER BY** 절 사용.

예: 사용자 조건은 거래일자로 한달조건 이나 ORDER BY 절은 고객번호, 계좌번호, 거래일자 인 경우

→ 1) 거래일자 인덱스로 한달을 엑세스(예:10만건) 한 후

2) 고객번호, 계좌번호, 거래일자 로 정렬

3) 정렬된 결과에서 첫 페이지 100건만 화면 표시.

➔ 비효율적인 9만 9천 9백건의 처리 발생.

**[논리 확인]**

전체 범위를 읽어서 정렬 조건별로 순번을 부여하여야만 결과집합이 완성되므로 전체 범위 처리로 수행된다.

```
SELECT * FROM (
SELECT RNUM01   ,
     NO      , HASH_KEY ,
     MOD_NO   , HASH_MOD_NO
FROM   ( SELECT /*+ INDEX(ANAF_TEST) */
            ROWNUM     RNUM01 ,
            NO      , HASH_KEY ,
            MOD_NO   , HASH_MOD_NO
       FROM   ANAF_TEST
       WHERE  HASH_KEY >= 1000000000
       AND    HASH_KEY <= 1999999999
       ORDER BY NO, HASH_KEY
     ) WHERE  ROWNUM    <= 200
) WHERE  RNUM01  BETWEEN 101   AND 200
```

```
Rows     Row Source Operation
-------  ---------------------------------------------
      0  STATEMENT
    100   VIEW
    200    COUNT STOPKEY
    200     VIEW
    200      SORT ORDER BY STOPKEY
  23166       COUNT
  23166        TABLE ACCESS BY INDEX ROWID ANAF_TEST
  23166         INDEX RANGE SCAN ANAF_TEST_X01
```

## 전체 범위 처리 되는 사례2

### ❑ GROUP BY 절, AGGREGATE 함수 사용.

예: 사용자 조건은 거래일자로 한달조건 이나 고객번호별로 거래건수 카운트 및 거래 금액 SUM

→ 1) 거래일자 인덱스로 한달을 엑세스(예:10만건) 한 후

2) 고객번호 별로 거래건수, 거래금액 합계 계산.

3) 정렬된 결과에서 첫 페이지 100건만 화면 표시.

➔ 고객번호순 101 번째 부터 나머지는 비효율.

**[논리 확인]**

전체 범위를 읽어서 그룹핑 조건별로 집계하여야만 결과집합이 완성되므로 전체 범위 처리로 수행된다.

```
SELECT * FROM (
SELECT ROWNUM    RNUM01   ,  MOD_NO   ,
     HASH_MOD_NO , CNT, MAX_KEY, MIN_KEY
FROM   (
      SELECT /*+ INDEX(ANAF_TEST) */
           HASH_MOD_NO, MOD_NO    ,
           COUNT(*)          CNT ,
           MAX(HASH_KEY)  MAX_KEY ,
           MIN(HASH_KEY)  MIN_KEY
      FROM   ANAF_TEST
      WHERE  HASH_KEY >= 1000000000
      AND    HASH_KEY <= 1999999999
      GROUP BY HASH_MOD_NO, MOD_NO
     ) WHERE  ROWNUM    <= 10
) WHERE  RNUM01  BETWEEN 6  AND 10 ;
```

```
Rows     Row Source Operation
-------  ---------------------------------------------------
      0  STATEMENT
      5   VIEW
     10    COUNT STOPKEY
     10     VIEW
     10      SORT GROUP BY STOPKEY
  23166       TABLE ACCESS BY INDEX ROWID ANAF_TEST
  23166        INDEX RANGE SCAN ANAF_TEST_X01
```

## 전체 범위 처리 되는 사례3

### ❑ SET 오퍼레이션 사용 사용.

예: 카드 승인 테이블에서 한달 조회 및 카드 취소 테이블에서 한달 조회 후 고객번호 별 정렬

→ 1) 카드 승인 테이블에서 한달 조건으로 대상 검색.

2) 카드 취소 테이블에서 한달 조건으로 대상 검색.

3) 카드 승인 결과와 카드 취소 결과를 고객번호 순으로 정렬.

4) 정렬된 결과에서 첫 페이지 100건만 화면 표시.

➔ 처음 100 건을 제외한 나머지는 비효율.

**[논리 확인]**

대상 집합들을 읽어서 SET 오퍼레이션 결과를 얻기 위해서는 대상 집합을 모두 처리해야만 하므로 전체 범위 처리로 수행 된다.

```
SELECT * FROM (
  SELECT RNUM01, NO, HASH_KEY, MOD_NO,HASH_MOD_NO
  FROM   ( SELECT ROWNUM     RNUM01 , NO ,
             HASH_KEY, MOD_NO , HASH_MOD_NO
         FROM   ANAF_TEST
         WHERE  HASH_KEY >= 1000000000
         AND    HASH_KEY <= 1499999999
         UNION
         SELECT ROWNUM    RNUM01 , NO ,
             HASH_KEY, MOD_NO , HASH_MOD_NO
         FROM   ANAF_TEST
         WHERE  HASH_KEY >= 1500000000
         AND    HASH_KEY <= 1999999999
        ) WHERE  ROWNUM    <= 200
) WHERE  RNUM01  BETWEEN 101   AND 200
```

```
Rows     Row Source Operation
-------  ---------------------------------------------------
      0  STATEMENT
      0   VIEW
    200    COUNT STOPKEY
    200     VIEW
    200     SORT UNIQUE STOPKEY
  23166      UNION-ALL
  11509       COUNT
  11509        TABLE ACCESS BY INDEX ROWID ANAF_TEST
  11509         INDEX RANGE SCAN ANAF_TEST_X01
  11657       COUNT
  11657        TABLE ACCESS BY INDEX ROWID ANAF_TEST
  11657         INDEX RANGE SCAN ANAF_TEST_X01
```

## 전체 범위 처리 되는 사례4

❑ **ROWNUM을 STOP KEY 가 아닌 COUNT/FILTER 로 사용.**

예: 페이지 처리시 ROWNUM 을 잘못 사용하는 경우

→ ROWNUM 은 항상 해당 쿼리 블럭의 조회 결과가 **나오는 순서대로 부여** 된다.

→ ROWNUM 은 항상 **'<='** 조건으로 사용해야만 **STOP KEY**로 동작한다.

→ ROWNUM 을 사용시 ROWNUM 부여시에 '<=' 조건(STOP KEY)을 부여해야만 올바른 페이지 처리가 가능 하다.

➔ 대부분의 ROWNUM 오류는 인라인뷰 안에서 ROWNUM 을 부여한 후, 인라인뷰 밖에서 STOPKEY 조건을 부여하는 경우 이다.

```
SELECT * FROM (
  SELECT RNUM01   ,
       NO      , HASH_KEY ,
       MOD_NO   , HASH_MOD_NO
  FROM   ( SELECT /*+ INDEX(ANAF_TEST) */
             ROWNUM     RNUM01 ,
             NO      , HASH_KEY ,
             MOD_NO   , HASH_MOD_NO
         FROM   ANAF_TEST
         WHERE  HASH_KEY >= 1000000000
         AND    HASH_KEY <= 1999999999
       )
) WHERE  RNUM01  BETWEEN 101   AND 200 ;
```

```
Rows     Row Source Operation
-------  ---------------------------------------------------
      0  STATEMENT
    100   VIEW
  23166    COUNT
  23166     TABLE ACCESS BY INDEX ROWID ANAF_TEST
  23166      INDEX RANGE SCAN ANAF_TEST_X01
```

**[논리 확인]**

별다른 정렬 조건이 없다면 인라인뷰 외부에서 ROWNUM STOP KEY 를 지정하면 해결 가능하다.

## 전체 범위 처리 되는 사례5

- 핵심 집합이 아닌 참조성 조인 이나 코드성 컬럼을 FROM절 집합으로 처리하는 경우.
- 핵심 집합 및 조건 SET.

  **사용자가 기술한 조건을** 처리하는 집합과,

  정렬조건이 존재하는 경우 **사용자 기술조건에서 정렬조건 까지의 조인 조건과,**

  결과 건수에 크게 영향을 주는 **중요 체크 조건.**
- 참조성 집합

  집합이 **증가하지 않는 아웃터 조인 집합(+)** 과,

  PK UNIQUE INDEX 로 조인되어 **명칭 등을 가져오는 코드성 조인** 과,

  최종 SELECT 절에서만 사용되는 **출력용 컬럼** 들.
- 카운트 SQL 은 핵심 집합만으로 처리되어야 하며, 참조성 집합의 처리가 발생된다면 모두 비효율 이다.
- 페이지 SQL 은 최종 페이지의 결과 nn건(예:20건)에 대해서만 참조성 집합의 엑세스가 발생되어야 하며, 전체에 대해서 수행되고 있다면 모두 비효율 이다.

## 전체 범위 처리 되는 사례5

```
 15  VIEW
186  COUNT
186   VIEW
186    SORT ORDER BY
186     FILTER
186      NESTED LOOPS
186       NESTED LOOPS OUTER
186        NESTED LOOPS OUTER
...         .....
186           NESTED LOOPS OUTER
186            NESTED LOOPS
186             NESTED LOOPS
186              TABLE ACCESS BY INDEX ROWID BCOTT
186               INDEX RANGE SCAN BCOTTASK_MST_I
186              TABLE ACCESS BY INDEX ROWID BMISA
186               INDEX UNIQUE SCAN BMISAGCY_PK
186             INDEX RANGE SCAN BZZZORG_IX02
186            TABLE ACCESS BY INDEX ROWID BMISTR
186             INDEX UNIQUE SCAN BMISTRM_OFFC_PK
186           TABLE ACCESS BY INDEX ROWID BCORC
186            INDEX UNIQUE SCAN BCORCONT_SVC_P
186          INDEX RANGE SCAN BCCUCUST_IX09
186         TABLE ACCESS BY INDEX ROWID BCORCO
186          INDEX UNIQUE SCAN BCORCONT_SBC_PK
184        INDEX RANGE SCAN BZZZMAPNG_CONT_IX
162       TABLE ACCESS BY INDEX ROWID BCOTPDA
162        INDEX UNIQUE SCAN BCOTPDA_PROC_INFO
186       TABLE ACCESS BY INDEX ROWID BCOTTRBL
186        INDEX UNIQUE SCAN BCOTTRBL_MST_PK
```

```
select * from (select inner_temp.* , rownum as lafindex from (
  SELECT /*+ LEADING(A) USE_NL(A C D I J L) */
       A.CUST_NO     , D.ORG_NM     ,
       G.TASK_NO     , I.WIRE_PC_CNT ,
       NVL(J.WLAN_CARD_OFFER_CNT, '0') WLAN_CARD_OFFER_CNT
 FROM   BCOTTASK_MST     A, BCOTTRBL_MST   B, BMISAGCY     C, BZZZORG D,
       BCOTPDA_PROC_INFO G, BMISTRM_OFFC   H, BCORCONT_SVC I,
       BCORCONT_SBC      J, BZZZMAPNG_CONT K, BCCUCUST     L
  WHERE A.TASK_NO             = B.TASK_NO
  AND   A.ASSGN_UNDTKR_OA_CD    = C.AGCY_CD
  AND   C.MNG_ORG_CD            = D.ORG_CD
  AND   A.TRM_OFFC_NO           = H.TRM_OFFC_NO(+)
  AND   A.TASK_NO             = G.TASK_NO(+)
  AND   A.SVC_LDGR_NO           = I.SVC_LDGR_NO
  AND   A.SBC_CONT_NO           = J.SBC_CONT_NO
  AND   A.SBC_CONT_NO           = K.SBC_CONT_NO(+)
  AND   A.CUST_NO             = L.CUST_NO
  AND   A.TASK_TYP            = 'TT'
  AND   A.ASSGN_UNDTKR_ID      IS NOT NULL
  AND   A.TASK_ST_TYP           = :1
  AND   A.ASSGN_UNDTKR_OA_CD    = :2
  AND   A.VIST_PREARNGE_YMD_TS >= :3||'000000'
  AND   A.VIST_PREARNGE_YMD_TS <  TO_CHAR(TO_DATE(:4,'yyyymmdd')+1,'yyyymmdd')
 ORDER BY B.LAST_DNN_RCP_SEQ DESC, A.VIST_PREARNGE_YMD_TS DESC
) inner_temp where rownum  <= :6 ) where lafindex >= :5
```

## ROWNUM 부여 및 STOP KEY 처리

예)

```
select * from (select  inner_temp.*, ROWNUM as lafindex  from (
SELECT
      A.CLAIM_NO     ,A.SBC_CUST_NO,B.SBC_CONT_NO
      ...
      ,F.CSALES_CD   ,C.PROD_CD   ,C.AGREE_TYP
 FROM  TB_BFSCCLAIM_MST A
      ,TB_BFSCCNSL B     ,TB_BCORCONT_SVC C
      ,TB_BMISAGCY D     ,TB_BMRSCSALES_CD F
      ,TB_BFSCCNSL_CD G
 WHERE A.CNSL_NO           = B.CNSL_NO
 AND   B.CNSL_SML_CLS_TYP   = G.COMM_CD_VAL
 AND   G.CLAIM_TASK_PUB_TYP = '2'
 AND   A.SVC_LDGR_NO        = C.SVC_LDGR_NO
 AND   A.PROC_AGCY_CD       = D.AGCY_CD
 AND   A.PROC_AGCY_CD       = F.SALES_PNT_NO
 AND   A.RCP_YMD          >= :1
 AND   A.RCP_YMD          <= :2
) inner_temp  where ROWNUM <= :11 ) where lafindex between :12 and :13
```

**[작업예제]**

```
SELECT * FROM (
SELECT RNUM01   ,
     NO      , HASH_KEY ,
     MOD_NO   , HASH_MOD_NO
FROM   ( SELECT /*+ INDEX(ANAF_TEST) */
           ROWNUM     RNUM01 ,
           CEIL(ROWNUM/100)  PNUM01 ,
           NO      , HASH_KEY ,
           MOD_NO   , HASH_MOD_NO
       FROM   ANAF_TEST
       WHERE  HASH_KEY >= 1000000000
       AND    HASH_KEY <= 1999999999
     ) WHERE  ROWNUM    <= 200
) WHERE  RNUM01  BETWEEN 101   AND 200 ;
```

## ROW_NUMBER( ) OVER ( ) 함수 방식 사용 불가

```
예)
select * from (
SELECT A.CLAIM_NO      ,A.SBC_CUST_NO ,B.SBC_CONT_NO
       ...
       , ROW_NUMBER() OVER ( ORDER BY A.RCP_YMD ) lafindex
       ,F.CSALES_CD   ,C.PROD_CD   ,C.AGREE_TYP
  FROM  TB_BFSCCLAIM_MST A
       ,TB_BFSCCNSL B      ,TB_BCORCONT_SVC C
       ,TB_BMISAGCY D      ,TB_BMRSCSALES_CD F
       ,TB_BFSCCNSL_CD G
  WHERE A.CNSL_NO            = B.CNSL_NO
  AND   B.CNSL_SML_CLS_TYP    = G.COMM_CD_VAL
  AND   G.CLAIM_TASK_PUB_TYP  = '2'
  AND   A.SVC_LDGR_NO         = C.SVC_LDGR_NO
  AND   A.PROC_AGCY_CD        = D.AGCY_CD
  AND   A.PROC_AGCY_CD        = F.SALES_PNT_NO
  AND   A.RCP_YMD           >= :1
  AND   A.RCP_YMD           <= :2
) where lafindex  between :12 and :13
```

**[문제점]**

→ 상수 조건 으로 페이지 조건 기술시에는 STOP KEY 로 인식되어 아무 문제 없이 정상 사용 가능.

→ 바인드 변수조건 으로 페이지 조건 기술시 COUNT/FILTER로 인식되어 전체 범위로 처리, 페이지 처리 사용 불가.

## COUNT(*) OVER( ) 함수 방식 사용 불가

예)

```
select * from (
SELECT A.CLAIM_NO    ,A.SBC_CUST_NO ,B.SBC_CONT_NO
        ...
        , ROW_NUMBER( ) OVER ( ORDER BY A.RCP_YMD ) lafindex
        , COUNT(*) OVER ( ) tot_cnt
        ,F.CSALES_CD    ,C.PROD_CD   ,C.AGREE_TYP
 FROM    TB_BFSCCLAIM_MST A
        ,TB_BFSCCNSL B    ,TB_BCORCONT_SVC C
        ,TB_BMISAGCY D    ,TB_BMRSCSALES_CD F
        ,TB_BFSCCNSL_CD G
 WHERE   A.CNSL_NO          = B.CNSL_NO
 AND    B.CNSL_SML_CLS_TYP   = G.COMM_CD_VAL
 AND    G.CLAIM_TASK_PUB_TYP = '2'
 AND    A.SVC_LDGR_NO       = C.SVC_LDGR_NO
 AND    A.PROC_AGCY_CD      = D.AGCY_CD
 AND    A.PROC_AGCY_CD      = F.SALES_PNT_NO
 AND    A.RCP_YMD          >= :1   AND A.RCP_YMD       <= :2
) where lafindex  between :12 and :13
```

**[문제점]**

→ COUNT(*) OVER () 사용시 **무조건 전체 범위로 처리** 된다.

→ 첫 페이지 수행시 별도의 카운트 쿼리가 불필요한 장점은 있으나, 두번째 페이지부터는 불필요하게 전체 범위를 처리하게 되는 비효율 발생한다.

**[문제점]**

→ 카운트를 모든 페이지에서 매번 구하게 되며 **매번 전체 범위로 처리** 된다.

→ 카운트는 매 조회의 **첫번째 페이지 에서만** 구해야 하며, 나머지 페이지는 조회건수를 전달해서 처리해야만 한다.

## 단순 페이지 분류 Analytic 함수 방식 사용 불가

예)

```
select * from (
SELECT A.CLAIM_NO   ,A.SBC_CUST_NO ,B.SBC_CONT_NO
        ...
        , CEIL( ROWNUM / :페이지목록수 )  lafindex
        , F.CSALES_CD   ,C.PROD_CD ,C.AGREE_TYP
 FROM    TB_BFSCCLAIM_MST A
        ,TB_BFSCCNSL B    ,TB_BCORCONT_SVC C
        ,TB_BMISAGCY D    ,TB_BMRSCSALES_CD F
        ,TB_BFSCCNSL_CD G
 WHERE A.CNSL_NO          = B.CNSL_NO
 AND   B.CNSL_SML_CLS_TYP   = G.COMM_CD_VAL
 AND   G.CLAIM_TASK_PUB_TYP = '2'
 AND   A.SVC_LDGR_NO       = C.SVC_LDGR_NO
 AND   A.PROC_AGCY_CD      = D.AGCY_CD
 AND   A.PROC_AGCY_CD      = F.SALES_PNT_NO
 AND   A.RCP_YMD          >= :1
 AND   A.RCP_YMD          <= :2
) where lafindex  = :14
```

**[문제점]**

→ STOP KEY 가 지정되지 않아서 전체대상에 대해서 CEIL() 함수를 수행 후 해당 페이지만을 가져오고 있다.

➔별다른 ORDERING 조건이 없다면 인라인뷰 밖에서 ROWNUM 조건기술 처리로 해결 가능 하다.

# 페이지 처리 SQL 엑세스 비효율 유형

❑ **페이지 처리 SQL 엑세스 비효율 유형.**

➔ 해당 페이지 시작조건에서 시작하지 못하고 누적 처리하는 엑세스 조건.

➔ 사용자 조건과 다른 정렬조건을 처리하기 위해 전체건에 대한 테이블 엑세스 처리.

엑세스 유형1 : 인덱스 → 테이블 → Order By & Group By ...

엑세스 유형2 : Outer Index → Outer Table → Inner Index → Inner Table → ...

→ Filter SubQuery → Order By & Group By ...

엑세스 유형3 : Main Table → 정합성 체크 Table(s) → 코드성 명칭 Table(s)

→ Order By & Group By ...

→ 붉은 표시부분에서 비효율이 발생하게 되며, 인덱스/테이블 엑세스 분리, 인덱스 선처리, ROWID 엑세스 기법 등을 사용하여 개선하게 된다.

❑ **엑세스 단계별로 엑세스 오브젝트와 처리범위, 건수를 확인하면 비효율을 확인할 수 있다.**

**→ 각 단계에서 최적의 엑세스인지, 불필요한 엑세스가 존재하는지 확인 가능하다.**

**예)** 인덱스 → 테이블 엑세스 에서 해당 페이지에 보여줄 목록(예:페이지당 20라인)이외의 테이블 엑세스는 비효율이다.

20 라인 화면의 2 페이지를 보여줄 때 21 번째부터 40 번째 라인만 테이블 엑세스가 발생하여야 비효율이 없다.

❑ **페이지 처리의 구성.**

→ 원 SQL ≠ 페이지 처리 SQL.

→ 페이지 처리 **=** 카운트 SQL + 페이지 SQL.

→ 정렬포함 SQL **Or** 정렬 미포함 SQL.

→ 페이지 순서 결정 집합 **+** 출력용 참조 집합.

❑ **페이지 처리 SQL 성능 참고 가이드.**

→ 페이지 처리의 핵심은 **페이지 결과SET의 순서를 결정하는 것** 이다.

→ 최소의 자원(인덱스 엑세스)으로 **결과SET 순서를 확정** 해야 한다.

→ ROWNUM 의 올바른 이해와 기술로 STOP KEY 처리를 통해 결과를 제한 한다.

→ 결과 집합의 유일성(결과 건수)을 저해하지 않는 조인 및 속성은 FROM 절 및 인라인뷰 내에서 제거, Scalar SubQuery 로 유도 한다.\

❑ 카운트 SQL

→ 카운트 쿼리는 전체 처리 범위를 엑세스 하므로 **항상 1 페이지 에서만 수행** 해야 하며, 나머지 페이지에서는 조회건수를 페이지 이동시 전달, 혹은 각 클라이언트 세션에서 유지 해야만 한다.

❑ 카운트 SQL 처리 가이드.

→ 카운트 SQL을 별도로 유지하거나 페이지SQL 외부에서 인라인뷰로 묶어 카운트를 구한다.

→ 개별 업무, 개별 SQL 단위로 수행하므로 아래의 3가지 중 하나로 구성된다.

카운트SQL과 페이지SQL로 구성.

첫 페이지SQL 과 이후 페이지SQL 로 구성.

페이지SQL 에 카운트 인라인뷰, 페이지 인라인뷰 를 씌워서 수행되게 구성.

➔ **카운트SQL 과 페이지SQL로 구분** 관리 한다.

→ 카운트SQL 사용은 최소한 집합(가능한 인덱스)으로 최적의 엑세스(핵심 업무 집합 등)만으로 구성되게 해야 하며, 페이지 SQL은 카운트 SQL에 정렬조건 과 SELECT 절의 컬럼만 추가 처리해 주면 대부분의 경우 최적화로 구현 된다.

❑ **최적화된 카운트 SQL 은 최적화된 페이지 SQL 로 변환 가능하다.**

## 성능 개선 가이드 1

❑ **페이지 처리 SQL 성능 개선 가이드.(카운트SQL과 페이지SQL에 동일)**

① 결과 집합의 순서를 결정하는 집합(조인 및 정렬조건 포함)이 존재하면 이를 인라인뷰로 분리 한다.

② ①번 인라인뷰 밖에서 참조용 집합(테이블 엑세스 및 참조성 조인 집합)을 조인 및 스칼라 서브쿼리 로 연결 한다.

③ ②번 결과 외부에서 COUNT(*) 추가. **← 카운트 쿼리 완성.**

④ ②번 결과 외부에서 ROWNUM 부여 및 'ROWNUM <= ' 조건으로 STOP KEY 를 지정하여 해당 페이지 까지 결과를 추출한다. **← 페이지 쿼리 완성.**

❑ **페이지SQL ➔ 카운트SQL 변환 및 검증 가이드.**

① 완성된 페이지 처리 SQL 에서 SELECT 절의 컬럼을 모두 제거한 후 상수 1 을 기술 한다.

② ORDER BY 절과 ROWNUM STOP KEY 처리를 위한 인라인뷰 및 조건을 제거한다.

③ SQL 트레이스 수집(확인) 후 결과집합이 유지되는 조인집합은 제거한다.

❑ **페이지 처리 성능 확인(실행계획 확인) .**

→ 카운트 쿼리로 수행시 가능한 인덱스로만 수행되게 한다.

→ 정렬 조건에 의해 전체범위 처리로 수행시 정렬에 필요한 집합을 인라인뷰로 먼저 수행하여 정렬 시킨다.

→ 정렬 이후의 참조성 집합은 스칼라 서브쿼리로 수행한다.(카운트 쿼리에서 보이지 않는다)

## 성능 개선 가이드 2

### ❑ 페이지 처리 SQL Advanced Guide

① 페이지 처리는 사용자 조건에 대한 **인덱스 엑세스를 기반** 으로 한다.

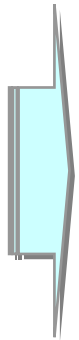
② 모든 사용자 조건을 인덱스에 포함 시킬 수 없으므로 순서 및 결과SET에 **핵심 속성을 인덱스로 구성** 하고, 낮은 변별력 속성은 **인라인 뷰 외부에서 테이블 엑세스시 체크** 한다.

③ **스칼라 서브쿼리를 최대한 활용** 하며, 스칼라 서브쿼리는 **최적 엑세스로 처리되게 인덱스를 구성**.

④ 논리적인 최적의 엑세스가 대부분 실제 물리적으로도 최적의 엑세스로 처리 된다.

➔ 필요시 **인덱스/테이블 엑세스 분리, 인덱스 선처리** 기법 등을 활용 한다.

```
SELECT RNUM01 ,
     NO    , HASH_KEY , MOD_NO  , HASH_MOD_NO
FROM   ( SELECT /*+ INDEX_DESC(ANAF_TEST) */
           ROWNUM     RNUM01 ,
           NO       , HASH_KEY ,
           MOD_NO   , HASH_MOD_NO
       FROM   ANAF_TEST
       WHERE  HASH_KEY >= 1000000000
       AND    HASH_KEY <= 1999999999
       ORDER BY NO, HASH_KEY ) T
WHERE  RNUM01    <= :END1
AND    RNUM01    >= :START1
```



```
SELECT /*+ ORDERED USE_NL(S T) */ S.RNUM01,
     S.NO, S.HASH_KEY,T.MOD_NO, T.HASH_MOD_NO
FROM   (
  SELECT ROWNUM   RNUM01, RID, NO, HASH_KEY
  FROM   ( SELECT ROWID     RID     ,
             NO       , HASH_KEY
         FROM   ANAF_TEST
         WHERE  HASH_KEY >= 1000000000
         AND    HASH_KEY <= 1999999999
         ORDER BY NO, HASH_KEY
        ) L WHERE ROWNUM  <= 200
     ) S , ANAF_TEST T
WHERE  S.RNUM01   >= 101
AND    S.RID      = T.ROWID
```

## 성능 개선 가이드 2

❑ 페이지 처리 **SQL Advanced Guide** 예제 확인)

| Call | Count | CPU Time | Elapsed Time | Disk | Query | Current | Rows |
|---|---|---|---|---|---|---|---|
| Total | 14 | 0.140 | 0.141 | 0 | 23181 | 0 | 100 |

| Rows | Row Source Operation |
|---|---|
| 0 | STATEMENT |
| 100 | FILTER |
| 100 | VIEW |
| 23166 | SORT ORDER BY |
| 23166 | COUNT |
| 23166 | TABLE ACCESS BY INDEX ROWID ANAF_TEST |
| 23166 | INDEX RANGE SCAN ANAF_TEST_X01 |

| Call | Count | CPU Time | Elapsed Time | Disk | Query | Current | Rows |
|---|---|---|---|---|---|---|---|
| Total | 13 | 0.020 | 0.019 | 0 | 176 | 0 | 100 |

| Rows | Row Source Operation |
|---|---|
| 0 | STATEMENT |
| 100 | NESTED LOOPS |
| 100 | VIEW |
| 200 | COUNT STOPKEY |
| 200 | VIEW |
| 200 | SORT ORDER BY STOPKEY |
| 23166 | INDEX RANGE SCAN ANAF_TEST_X01 |
| 100 | TABLE ACCESS BY USER ROWID ANAF_TEST |

## 페이지 처리 SQL 가이드

❑ **페이지 처리 SQL 작성 가이드.**

➔ 페이지 처리는 결과 집합의 순서를 인덱스로 맞춰 해당 페이지까지 만의 엑세스 처리가 목표 이며,

➔ 결과 집합의 순서를 보장하기 위한 비효율을 최소화 하는 것이 방법 이다.

➔ 최적화된 카운트SQL의 처리가 페이지 처리의 최적화 지표 이다.

❑ **불필요한 사용자 정렬조건은 제거 가능한지 업무 확인이 필요하다.**

❑ **정렬 집합 먼저 처리.**

인덱스를 이용한 사용자 엑세스 조건과 다른 정렬 조건을 가졌다면 관련 집합을 먼저 처리한다.
인라인 뷰 내에서 별도의 엑세스(조인 및 ORDER BY 수행)로 처리한다.
필요시 인덱스와 테이블 엑세스를 분리한다.

❑ **결과 집합의 유일성(결과 건수)에 영향이 없는 참조성 컬럼/조인은 제거한다.**

결과 집합에 영향이 없는 참조성, 코드성 조인(SQL Trace 시 Rows에 변경이 없는 조인으로 주로 PK Unique Index 엑세스 유형) 및 컬럼은 **스칼라 서브쿼리 로 변경** 한다.

❑ **엑세스에 맞는 적정한 인덱스를 구성 한다.**

❑ **전체 범위 처리를 가능한 제거한다.**

( SET 오퍼레이션, DISTINCT, GROUP BY, AGGREGATE 함수 등등)

## 페이지 처리 SQL 가이드 예제

```
select * from (select inner_temp.* , rownum as lafindex from (
   SELECT /*+ LEADING(A) USE_NL(A C D I J L) */
        A.CUST_NO     , D.ORG_NM      ,
        G.TASK_NO     , I.WIRE_PC_CNT , ... ,
        NVL(J.WLAN_CARD_OFFER_CNT, '0') WLAN_CARD_OFFER_CNT
 FROM    BCOTTASK_MST      A, BCOTTRBL_MST   B, BMISAGCY     C, BZZZORG D,
        BCOTPDA_PROC_INFO G, BMISTRM_OFFC   H, BCORCONT_SVC I,
        BCORCONT_SBC      J, BZZZMAPNG_CONT K, BCCUCUST     L
   WHERE A.TASK_NO              = B.TASK_NO
   AND   A.ASSGN_UNDTKR_OA_CD    = C.AGCY_CD
   AND   C.MNG_ORG_CD           = D.ORG_CD
   AND   A.TRM_OFFC_NO          = H.TRM_OFFC_NO(+)
   AND   A.TASK_NO              = G.TASK_NO(+)
   AND   A.SVC_LDGR_NO          = I.SVC_LDGR_NO
   AND   A.SBC_CONT_NO          = J.SBC_CONT_NO
   AND   A.SBC_CONT_NO          = K.SBC_CONT_NO(+)
   AND   A.CUST_NO              = L.CUST_NO
   AND   A.TASK_TYP             = 'TT'
   AND   A.ASSGN_UNDTKR_ID      IS NOT NULL
   AND   A.TASK_ST_TYP          = :1
   AND   A.ASSGN_UNDTKR_OA_CD    = :2
   AND   A.VIST_PREARNGE_YMD_TS >= :3||'000000'
   AND   A.VIST_PREARNGE_YMD_TS <  TO_CHAR(TO_DATE(:4,'yyyymmdd')+1,'yyyymmdd')
 ORDER BY B.LAST_DNN_RCP_SEQ DESC, A.VIST_PREARNGE_YMD_TS DESC
) inner_temp where rownum  <= :6 ) where lafindex >= :5
```

## 페이지 처리 SQL 가이드 예제

```
Rows  Row Source Operation
 15  FILTER
 15   VIEW
186    COUNT
186     VIEW
186      SORT ORDER BY
186       FILTER
186        NESTED LOOPS
186         NESTED LOOPS OUTER
186          NESTED LOOPS OUTER
...           .....
186             NESTED LOOPS OUTER
186              NESTED LOOPS
186               NESTED LOOPS
186                TABLE ACCESS BY INDEX ROWID BCOTTASK_MST
186                 INDEX RANGE SCAN BCOTTASK_MST_IX18
186                TABLE ACCESS BY INDEX ROWID BMISAGCY
186                 INDEX UNIQUE SCAN BMISAGCY_PK
186              INDEX RANGE SCAN BZZZORG_IX02
186             TABLE ACCESS BY INDEX ROWID BMISTRM_OFFC
186              INDEX UNIQUE SCAN BMISTRM_OFFC_PK
186            TABLE ACCESS BY INDEX ROWID BCORCONT_SVC
186             INDEX UNIQUE SCAN BCORCONT_SVC_PK
186           INDEX RANGE SCAN BCCUCUST_IX09
186          TABLE ACCESS BY INDEX ROWID BCORCONT_SBC
186           INDEX UNIQUE SCAN BCORCONT_SBC_PK
184         INDEX RANGE SCAN BZZZMAPNG_CONT_IX01
162        TABLE ACCESS BY INDEX ROWID BCOTPDA_PROC_INFO
162         INDEX UNIQUE SCAN BCOTPDA_PROC_INFO_PK
186        TABLE ACCESS BY INDEX ROWID BCOTTRBL_MST
186         INDEX UNIQUE SCAN BCOTTRBL_MST_PK
```

```
Rows  Row Source Operation
 15  VIEW
 15   COUNT STOPKEY
 15    NESTED LOOPS
 15     NESTED LOOPS
 15      NESTED LOOPS
 15       NESTED LOOPS
 15        NESTED LOOPS
 15         VIEW
 15          SORT ORDER BY
186           FILTER
186            NESTED LOOPS
186             TABLE ACCESS BY INDEX ROWID BCOTTASK_MST
186              INDEX RANGE SCAN BCOTTASK_MST_IX18
186             TABLE ACCESS BY INDEX ROWID BCOTTRBL_MST
186              INDEX UNIQUE SCAN BCOTTRBL_MST_PK
 15          INDEX RANGE SCAN BCCUCUST_IX09
 15         TABLE ACCESS BY INDEX ROWID BMISAGCY
 15          INDEX UNIQUE SCAN BMISAGCY_PK
 15        INDEX RANGE SCAN BZZZORG_IX02
 15       TABLE ACCESS BY INDEX ROWID BCORCONT_SBC
 15        INDEX UNIQUE SCAN BCORCONT_SBC_PK
 15      TABLE ACCESS BY INDEX ROWID BCORCONT_SVC
 15       INDEX UNIQUE SCAN BCORCONT_SVC_PK
```

| call | count | cpu | elapsed | disk | query | rows |
|---|---|---|---|---|---|---|
| 개선전 | 3 | 0.80 | 7.91 | 1232 | 6168 | 15 |
| 개선후 | 3 | 0.22 | 1.59 | 282 | 1341 | 15 |

## 페이지 처리 SQL 가이드 예제

```
select * from (select inner_temp.* , rownum as lafindex from (
  SELECT /*+ LEADING(A) USE_NL(A C D I J L) */
       A.CUST_NO , D.ORG_NM     , G.TASK_NO , I.WIRE_PC_CNT,
      ( SELECT H.TRM_OFFC_NM  FROM TB_BMISTRM_OFFC H  WHERE A.TRM_OFFC_NO
= H.TRM_OFFC_NO ) TRM_OFFC_NM , ...
  FROM  ( SELECT A.CUST_NO    ,A.SBC_CONT_NO ,A.TASK_NO    ,
            A.SVC_LDGR_NO,B.TRBL_ACT_CD ,B.TRBL_GD_TYP
        FROM   TB_BCOTTASK_MST A, TB_BCOTTRBL_MST B
        WHERE  A.TASK_NO             = B.TASK_NO
         AND   ASSGN_UNDTKR_ID       IS NOT NULL
         AND   A.TASK_ST_TYP         = :1
         AND   A.ASSGN_UNDTKR_OA_CD    = :2
         AND   A.VIST_PREARNGE_YMD_TS >= :3||'000000'
         AND   A.VIST_PREARNGE_YMD_TS <
           TO_CHAR(TO_DATE(:4,'yyyymmdd')+1,'yyyymmdd')
       ORDER BY B.LAST_DNN_RCP_SEQ DESC,
            A.VIST_PREARNGE_YMD_TS DESC
      ) A , BMISAGCY C, BZZZORG D, BCORCONT_SVC I, BCORCONT_SBC J, BCCUCUST L
  WHERE A.ASSGN_UNDTKR_OA_CD = C.AGCY_CD
  AND   C.MNG_ORG_CD        = D.ORG_CD
  AND   A.SVC_LDGR_NO       = I.SVC_LDGR_NO
  AND   A.SBC_CONT_NO       = J.SBC_CONT_NO
  AND   A.CUST_NO           = L.CUST_NO
 ) inner_temp where rownum  <= :6 ) where lafindex >= :5
```

```
Rows  Row Source Operation                                  .
  15  VIEW
  15   COUNT STOPKEY
  15    NESTED LOOPS
  15     NESTED LOOPS
  15      NESTED LOOPS
  15       NESTED LOOPS
  15        NESTED LOOPS
  15         VIEW
  15          SORT ORDER BY
 186           FILTER
 186            NESTED LOOPS
 186             TABLE ACCESS BY INDEX ROWID BCOTTASK_MST
 186              INDEX RANGE SCAN BCOTTASK_MST_IX18
 186             TABLE ACCESS BY INDEX ROWID BCOTTRBL_MST
 186              INDEX UNIQUE SCAN BCOTTRBL_MST_PK
  15          INDEX RANGE SCAN BCCUCUST_IX09
  15         TABLE ACCESS BY INDEX ROWID BMISAGCY
  15          INDEX UNIQUE SCAN BMISAGCY_PK
  15        INDEX RANGE SCAN BZZZORG_IX02
  15       TABLE ACCESS BY INDEX ROWID BCORCONT_SBC
  15        INDEX UNIQUE SCAN BCORCONT_SBC_PK
  15      TABLE ACCESS BY INDEX ROWID BCORCONT_SVC
  15       INDEX UNIQUE SCAN BCORCONT_SVC_PK
```

7. 성능개선 사례

- 잘 못 풀린 실행계획
- 컬럼 가공으로 인한 비효율
- DYNAMIC SQL
- 최적화되지 않은 COUNT QUERY
- 최소, 최대값(최초, 최종) 구하기
- 함수 수행횟수 비효율
- 빈번한 OCI CALL과 LOOP QUERY
- 부분범위처리
- 인덱스 부재
- 인덱스의 잘 못된 이해
- 불필요한 조인

# 7. 성능개선 사례

비효율 SQL

```
SELECT count(*)
  FROM r_waitfor
 WHERE (    cust_mng_num_friend=:b1
        and cust_mng_num_friend2=:b2)
    or (    cust_mng_num_friend=:b2
        and cust_mng_num_friend2=:b1)
```

비효율 실행계획

```
SELECT STATEMENT Optimizer=FIRST_ROWS
  SORT (AGGREGATE)
    INDEX (FAST FULL SCAN) OF PK_WAITFOR (UNIQUE)
```

개선 SQL

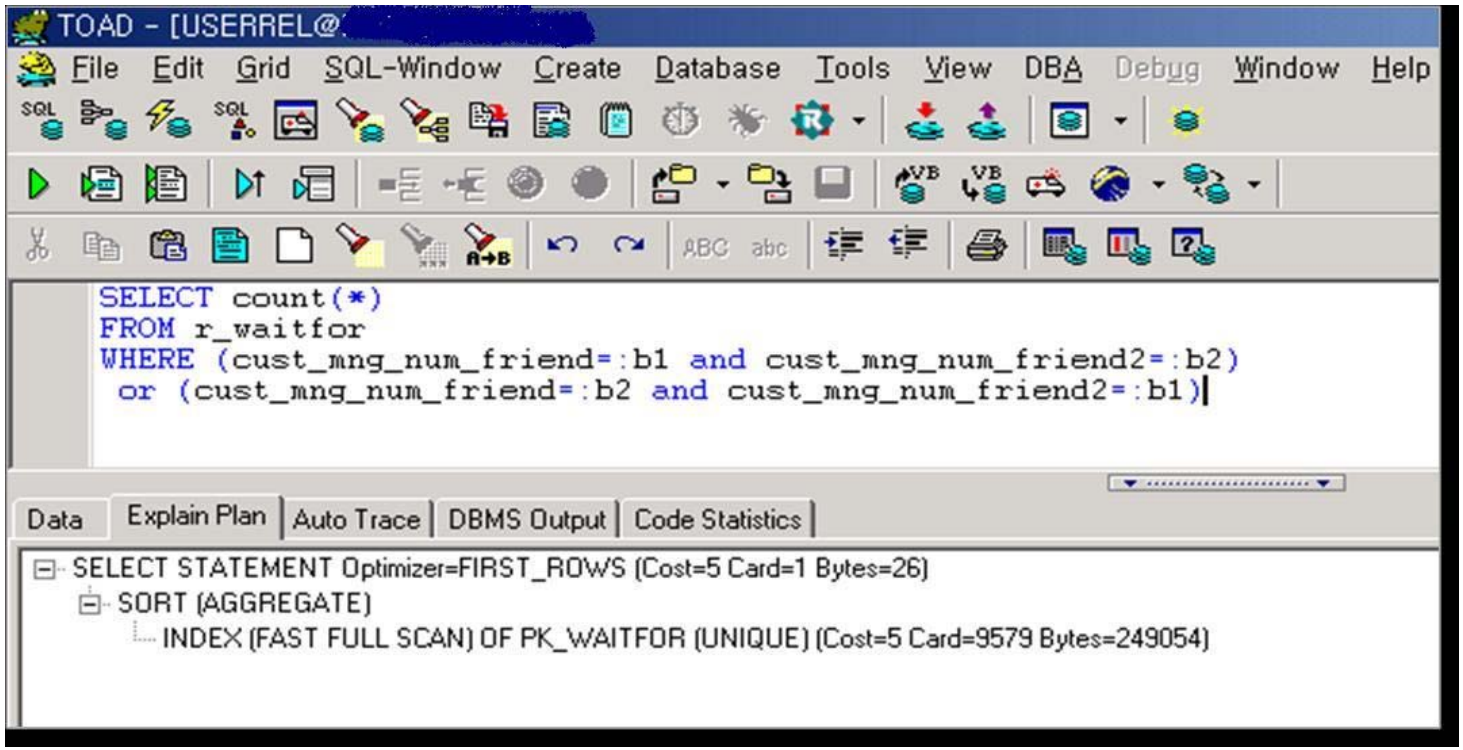
```
SELECT /*+ USE_CONCAT */ count(*)
  FROM r_waitfor
 WHERE (    cust_mng_num_friend=:b1
        and cust_mng_num_friend2=:b2)
    OR (    cust_mng_num_friend=:b2
        and cust_mng_num_friend2=:b1)
```

개선 실행계획

```
SELECT STATEMENT Optimizer=HINT: RULE
  SORT (AGGREGATE)
    CONCATENATION
      AND-EQUAL
        INDEX (RANGE SCAN) OF IDX_RWAITFOR_FRIEND (NON-UNIQUE)
        INDEX (RANGE SCAN) OF IDX_RWAITFOR_FRIEND2 (NON-UNIQUE)
      AND-EQUAL
        INDEX (RANGE SCAN) OF IDX_RWAITFOR_FRIEND (NON-UNIQUE)
        INDEX (RANGE SCAN) OF IDX_RWAITFOR_FRIEND2 (NON-UNIQUE)
```

## 주의사항

**SQL**을 수행 또는 적용하기 전 반드시 실행계획 확인을 습관화 해야만 문제를 유발하지 않을 수 있다.



# 컬럼 가공으로 인한 비효율

## 비효율 SQL

```
SELECT count(*)
  FROM R_MEMO a
 WHERE to_char(a.memo_regdate, 'YYYYMMDD')
         = to_char(sysdate, 'YYYYMMDD')
```

## 비효율 실행계획

```
SELECT STATEMENT Optimizer=FIRST_ROWS
  SORT (AGGREGATE)
    TABLE ACCESS (FULL) OF R_MEMO
```

개선 **SQL**

```
SELECT count(*)
  FROM R_MEMO a
 WHERE a.memo_regdate between trunc(sysdate)
                          and sysdate
```

개선 실행계획

```
SELECT STATEMENT Optimizer=FIRST_ROWS
  SORT (AGGREGATE)
    FILTER
      INDEX (RANGE SCAN) OF IDX_LIST_RMEMO (NON-UNIQUE)
```

## 유사한 사례 - 1

컬럼 데이터 타입과 다른 타입의 상수값 제공으로 컬럼에 내부변형이 발생

```
SELECT cust_mng_num_friend
  FROM r_waitfor
 WHERE cust_mng_num = :3
   and waitfor_type = '0'
   and waitfor_result1 = 0 -- 컬럼이 Char Type
   and waitfor_result2 = 0 -- 컬럼이 Char Type
```

**개선 SQL**

```
SELECT cust_mng_num_friend
  FROM r_waitfor
 WHERE cust_mng_num = :3
   and waitfor_type = '0'
   and waitfor_result1 =  '0'  -- 컬럼과 동일한 Type 상수값
   and waitfor_result2 =  '0'  -- 컬럼과 동일한 Type 상수값
```

## 유사한 사례 - 2

컬럼 데이터 타입과 다른 타입의 변수 바인딩

```
SELECT schoolName
  FROM tblCodeSchool
 WHERE schoolcd = :1 -- 컬럼이 Char Type, 변수는 숫자 타입
                     -- 타입이 서로 달라 컬럼에 내부변형이 일어남
```

비효율 실행계획

```
Rows     Row Source Operation
-------  ---------------------------------------
      1  TABLE ACCESS FULL TBLCODESCHOOL
```

## 유사한 사례 - 2

### 개선 SQL

```
SELECT schoolName
  FROM tblCodeSchool
 WHERE schoolcd = :1 -- 컬럼이 Char Type, 변수도 문자 타입
                     -- 타입이 같아 컬럼에 내부변형이
                     -- 일어나지 않음
```

### 개선 실행계획

```
Rows     Row Source Operation
-------  ---------------------------------------------------
      1  TABLE ACCESS BY INDEX ROWID TBLCODESCHOOL
      1   INDEX UNIQUE SCAN SCHOOLCD_PK
```

## Dynamic SQL 사용 사례

**사례 1**

```
 BEGIN SP_SET_RECOM_CONF
( 'emdeotja',  '2',  '35',  '38',  '055',  '',  '',  '');
  END;
```

**사례 2**

```
SELECT /*+ use_nl(a,c,d,b,e) ordered */
       count(a.cust_mng_num_friend)
  FROM R_WAITFOR a, R_PROFILE c, R_CARD d, TB_SVC_INFO b, TB_CUST_INFO e
 WHERE a.cust_mng_num_friend = :1
   and (a.waitfor_result2 in ('0', '1', '3', '4', '5', '6'))
   and (a.cust_mng_num = b.svc_inst_no)
   and (a.cust_mng_num = c.cust_mng_num)
   and (a.cust_mng_num = d.cust_mng_num)
   and (b.svc_inst_no = c.cust_mng_num)
   and (b.svc_inst_no = d.cust_mng_num)
   and (b.cust_mng_no = e.cust_mng_no)
   and (c.cust_mng_num = d.cust_mng_num)
   and (b.svc_id like 'kh91%')
```

주의사항

❑다양한 값으로 제공되는 모든 변수를 반드시 **Static SQL (Parameter Binding)**으로 처리
❑자주 사용되는 프로그램부터 우선 적용하면 개선 효과가 빨리 나타남

## 최적화 되지 않은 Count Query

데이터 추출 Query

```
SELECT *
  FROM (SELECT rownum rn, x.*
          FROM (
SELECT /*+ use_nl(a,c,d,b,e) ordered */
       a.cust_mng_num, a.cust_mng_num_friend, a.cust_mng_num_friend2,
       TO_CHAR(a.waitfor_regdate, 'YYYY-MM-DD') waitfor_regdate,
       TO_CHAR(a.waitfor_lastupdate2, 'YYYY-MM-DD') waitfor_lastupdate2,
       a.waitfor_type, a.waitfor_result2, a.waitfor_memo, b.svc_id cust_id,
       c.profile_gender, c.profile_age, d.card_openmode2, e.cust_nm cust_name
  FROM R_WAITFOR a, R_PROFILE c, R_CARD d, TB_SVC_INFO b, TB_CUST_INFO e
 WHERE a.cust_mng_num_friend = :1
   and (a.waitfor_result2 in ('0', '1', '3', '4', '5', '6'))
   and (a.cust_mng_num = b.svc_inst_no)  and (a.cust_mng_num = c.cust_mng_num)
   and (a.cust_mng_num = d.cust_mng_num) and (b.svc_inst_no = c.cust_mng_num)
   and (b.svc_inst_no = d.cust_mng_num)  and (b.cust_mng_no = e.cust_mng_no)
   and (c.cust_mng_num = d.cust_mng_num)
ORDER BY a.waitfor_lastupdate2 DESC) x)
 WHERE rn BETWEEN :2 AND :3
```

Count Query

```
SELECT /*+ use_nl(a,c,d,b,e) ordered */
       count(a.cust_mng_num_friend)
  FROM R_WAITFOR a, R_PROFILE c, R_CARD d, TB_SVC_INFO b, TB_CUST_INFO e
 WHERE a.cust_mng_num_friend = :1
   and (a.waitfor_result2 in ('0', '1', '3', '4', '5', '6'))
   and (a.cust_mng_num = b.svc_inst_no)   and (a.cust_mng_num = c.cust_mng_num)
   and (a.cust_mng_num = d.cust_mng_num)  and (b.svc_inst_no = c.cust_mng_num)
   and (b.svc_inst_no = d.cust_mng_num)   and (b.cust_mng_no = e.cust_mng_no)
   and (c.cust_mng_num = d.cust_mng_num)
```

## Count Query 개선 사례

### 개선 SQL

```
SELECT count(*)
  FROM R_WAITFOR a
 WHERE a.cust_mng_num_friend = :1
   and a.waitfor_result2 in ('0', '1', '3', '4', '5', '6')
```

### 개선 실행계획

```
SELECT STATEMENT Optimizer=FIRST_ROWS
  SORT (AGGREGATE)
    TABLE ACCESS (BY INDEX ROWID) OF R_WAITFOR
      INDEX (RANGE SCAN) OF IDX_RWAITFOR_FRIEND
```

### 주의사항

❑데이터 추출 쿼리에서 **Select** 컬럼을 **Count**문으로 대체하는 방법은 비효율을 발생시키는 주원인
❑**Count** 하고자 하는 작업의도에 최적화된 **Query**를 작성

## 최소값 구하기 비효율 사례

### 비효율 SQL

```
SELECT skin_id, skin_name, skin_url, skin_preview,
       skin_color1, skin_color2, skin_color3,
       skin_color4, skin_color5, skin_color6, skin_color7
  FROM raddr_skinref
 WHERE skin_order = (SELECT min(skin_order)
                       FROM raddr_skinref)
```

### 비효율 실행계획

```
SELECT STATEMENT Optimizer=FIRST_ROWS
  TABLE ACCESS (BY INDEX ROWID) OF RADDR_SKINREF
    INDEX (RANGE SCAN) OF IDX_SKIN_ORDER (NON-UNIQUE)
      SORT (AGGREGATE)
        INDEX (FULL SCAN (MIN/MAX)) OF IDX_SKIN_ORDER (NON-UNIQUE)
```

## 최소값 구하기 비효율 사례

### 개선 SQL

```
SELECT /*+ index(a IDX_SKIN_ORDER) */
       skin_id, skin_name, skin_url, skin_preview,
       skin_color1,skin_color2, skin_color3, skin_color4,
       skin_color5, skin_color6, skin_color7
  FROM raddr_skinref a
 WHERE rownum <= 1
```

### 개선 실행계획

```
SELECT STATEMENT Optimizer=FIRST_ROWS
  COUNT (STOPKEY)
    TABLE ACCESS (BY INDEX ROWID) OF RADDR_SKINREF
      INDEX (FULL SCAN) OF IDX_SKIN_ORDER (NON-UNIQUE)
```

### 주의사항

❑최소 또는 최초는 인덱스를 오름차순으로 읽어 조건에 맞는 첫번째 로우에서 멈춘다.
❑최대 또는 최종은 인덱스를 내림차순으로 읽어 조건에 맞는 첫번째 로우에서 멈춘다.
**(index_desc 힌트이용)**

## 불필요한 함수 사용 사례

### 비효율 SQL

```
SELECT a.cust_mng_num
      ,NVL(a.profile_cmpname,''),     NVL(a.profile_cmppart,'')
      ,NVL(a.profile_cmpposition,''), NVL(a.profile_mobileprefix,'')
      ,NVL(a.profile_mobilenum,'')
  FROM R_PROFILE a
 WHERE a.cust_mng_num=:b1
```

### 개선 SQL

```
SELECT a.cust_mng_num
      ,a.profile_cmpname,     a.profile_cmppart
      ,a.profile_cmpposition, a.profile_mobileprefix
      ,a.profile_mobilenum
  FROM R_PROFILE a
 WHERE a.cust_mng_num=:b1
```

### 주의사항

❑함수를 꼭 사용할 필요가 있을 때만 사용하고 무분별하게 남용하면 안됨

## 빈번한 OCI Call과 Loop Query

비효율 **SOURCE**

최대 **5**번, 최소 **1**번 **SQL** 수행

```
SELECT COUNT(*) INTO cnt /* 확인 */
FROM pims_cgroup
WHERE cust_mng_num =:NEW.SVC_INST_NO;

IF (cnt = 0 ) THEN /* 없으면 INSERT 4번 실행 */
          INSERT INTO pims_cgroup
           (cust_mng_num, gid, gname, create_date)
           VALUES (:NEW.SVC_INST_NO, 0, '기타', SYSDATE);
          INSERT INTO pims_cgroup
           (cust_mng_num, gid, gname, create_date)
           VALUES (:NEW.SVC_INST_NO, 1, '가족', SYSDATE);
          INSERT INTO pims_cgroup
           (cust_mng_num, gid, gname, create_date)
           VALUES (:NEW.SVC_INST_NO, 2, '친구', SYSDATE);
          INSERT INTO pims_cgroup
           (cust_mng_num, gid, gname, create_date)
           VALUES (:NEW.SVC_INST_NO, 3, '직장', SYSDATE);
END IF;
```

## 빈번한 OCI Call과 Loop Query

One SQL 통합으로 최대 1번 수행

개선 SQL

```
INSERT INTO userpims.pims_cgroup(cust_mng_num, gid, gname, create_date)
select *
from
          (select :NEW.SVC_INST_NO, 0, '기타', SYSDATE from dual
          union all select :NEW.SVC_INST_NO, 1, '가족', SYSDATE from dual
          union all select :NEW.SVC_INST_NO, 2, '친구', SYSDATE from dual
          union all select :NEW.SVC_INST_NO, 3, '직장', SYSDATE from dual)
where not exists
          (select '1'
          from userpims.pims_cgroup
          where cust_mng_num = :NEW.SVC_INST_NO);
```

개선 실행계획

```
INSERT STATEMENT Optimizer=FIRST_ROWS
  FILTER
    VIEW
      UNION-ALL
        TABLE ACCESS (FULL) OF DUAL
        TABLE ACCESS (FULL) OF DUAL
        TABLE ACCESS (FULL) OF DUAL
        TABLE ACCESS (FULL) OF DUAL
    INDEX (RANGE SCAN) OF PK_PIMS_CGROUP (UNIQUE)
```

**Loop Query**는 **OCI Call** 부하를 비롯한 여러 비효율을 유발하므로 사용을 자제 가능한 **One SQL**로 통합 할 것

## WEB 게시판 사례

문제 : 온라인 업무에서 가장 자주 사용되는 경우이면서 가장 많은 비효율을 가지고 있다.

원인 :

- ❑ **SQL**문 구사하는 방법의 부족
- ❑ 모든 조건에 만족하는 이 아니라 꼭 필요한 조건이 무엇인지의 전략적인 선택의 부족
- ❑ 인덱스 전략의 부족

## WEB 게시판 사례

### 개선전 SQL

**1|2|...다음(Navigation)**

```
SELECT count(*)
  FROM NC_ChatProfile
WHERE user_id not in ('a-002')
      AND client_mode = 'N'
      AND is_online = 'T'
      AND cust_name like '%김%'
```

페이지 리스트

```
SELECT rnum , user_id, cust_name
  FROM (SELECT rownum rnum , user_id,
               cust_name
          FROM NC_ChatProfile
         WHERE user_id not in ('a-002')
           AND client_mode = 'N'
           AND is_online = 'T'
           AND cust_name like '%김%'
           AND rownum <= 30
       ) where rnum between 16 and 30
```

### 개선후 SQL

```
SELECT RNUM, USER_ID, CUST_NAME
  FROM (SELECT ROWNUM RNUM, USER_ID, CUST_NAME
          FROM (SELECT /*+ INDEX(A NC_CHATPROFILE_IDX2) */
                       USER_ID, CUST_NAME
                  FROM NC_CHATPROFILE A
                 WHERE 'NEXT' = UPPER(:SW)
                   AND IS_ONLINE = 'T'
                   AND CLIENT_MODE = 'N'
                   AND CUST_NAME >= :INIT_C_NM
                   AND CUST_NAME LIKE :C_NM || '%') X
         WHERE ROWNUM <= (15*10)+1
        UNION ALL
        SELECT ((15 * 10) + 2 - ROWNUM) RNUM, USER_ID, CUST_NAME
          FROM (SELECT /*+ INDEX_DESC(A NC_CHATPROFILE_IDX2) */
                       FROM NC_CHATPROFILE A
                      WHERE  'PREV' = UPPER(:SW)
                      ..... ) X
         WHERE ROWNUM <= (15*10)+1) Y
 WHERE RNUM IN (1, 16, 31, 46, 61, 76, 91, 106, 121, 136, 151)
 ORDER BY RNUM
```

```
SELECT USER_ID, CUST_NAME
  FROM (SELECT /*+ INDEX(A NC_CHATPROFILE_IDX2) */
               USER_ID, CUST_NAME
          FROM NC_CHATPROFILE A
         WHERE IS_ONLINE = 'T'
           AND CLIENT_MODE = 'N'
           AND CUST_NAME > :INIT_C_NM
           AND CUST_NAME LIKE :C_NM || '%')
 WHERE ROWNUM <= 15
```

## 인덱스 부적절 사례

매 시간마다 현재 온라인 사용자 수 저장하는 **SQL**

```
INSERT INTO TKSTATCURUSERLOG ( statdate, mediatype, equipmenttype, value )
SELECT TO_CHAR(SYSDATE, 'YYYYMMDDHH24'), c.mediatype, c.equipmenttype, NVL(cnt, 0)
   FROM  ( SELECT mediatype, equipmenttype, COUNT(*) AS cnt
                  FROM TKAUTH
                 WHERE isonline='Y'
                 GROUP BY mediatype, equipmenttype
             ) t, TKCLIENTVER c
  WHERE t.mediatype(+)      =  c.mediatype
      AND t.equipmenttype(+) = c.equipmenttype;
```

소요시간 :
**3분 ~ 5분**

```
INSERT STATEMENT, GOAL = FIRST_ROWS
 HASH JOIN OUTER
  INDEX FAST FULL SCAN          PK_TKCLIENTVER
  VIEW
   SORT GROUP BY
    TABLE ACCESS FULL           TKAUTH
```

- ❑ 온라인 배치 작업으로 서비스 중에 **1**시간 간격으로 작업수행
- ❑ 하루 평균 데이터 **1**만 건 증가 추세
- ❑ 최대한 짧은 시간 내에 작업을 끝내지 않으면 장애 유발 가능

## 인덱스 부적절 사례

개선 **SQL**

- ❑ 처리속도를 극대화하기 위해 **Where**절 뿐만 아니라 **group by** 절까지 **index** 컬럼으로 구성하여 **index block**만 읽고 처리
- ❑ **Tkauth**에 인덱스 생성**(isonline+mediatype+equipmenttype)**
- ❑ 데이터가 증가하여도 **online**사용자수는 일정하므로 **index**의 일량은 항상 동일
- ❑ 처리속도 극대화로 온라인 배치작업으로 인한 장애 유발을 피함
- ❑ **SQL**변경 없음

소요시간 :
**1초 ~ 3초**

```
EXECUTE PLAN
----------------------------------------------------------------------
INSERT STATEMENT, GOAL = FIRST_ROWS

 MERGE JOIN OUTER
  INDEX FULL SCAN     PK_TKCLIENTVER
  SORT JOIN
   VIEW
    SORT GROUP BY
     INDEX RANGE SCAN          TKAUTH_ISONLINE_IDX
```

## 부적합한 인덱스의 사용

```
INSERT INTO TKSTATLOGINCNT
SELECT '01', TO_CHAR(SYSDATE - 1, 'YYYYMMDD'), c.mediatype, c.equipmenttype, NVL(cnt,0)
  FROM (SELECT /*+ index tkuserlog TKUSERLOG_CMN_IN_OUT_IDX) */
               mediatype, equipmenttype, COUNT(*) AS cnt
          FROM TKUSERLOG
         WHERE cust_mng_num > 0
           AND logindate LIKE TO_CHAR(SYSDATE - 1, 'YYYYMMDD') || '%'
           AND logoutdate < TO_CHAR(SYSDATE, 'YYYYMMDD') || '000000'
         GROUP BY mediatype, equipmenttype
       ) a, TKCLIENTVER c
 WHERE a.mediatype    = c.mediatype
   AND a.equipmenttype = c.equipmenttype
```

전체:5천만건

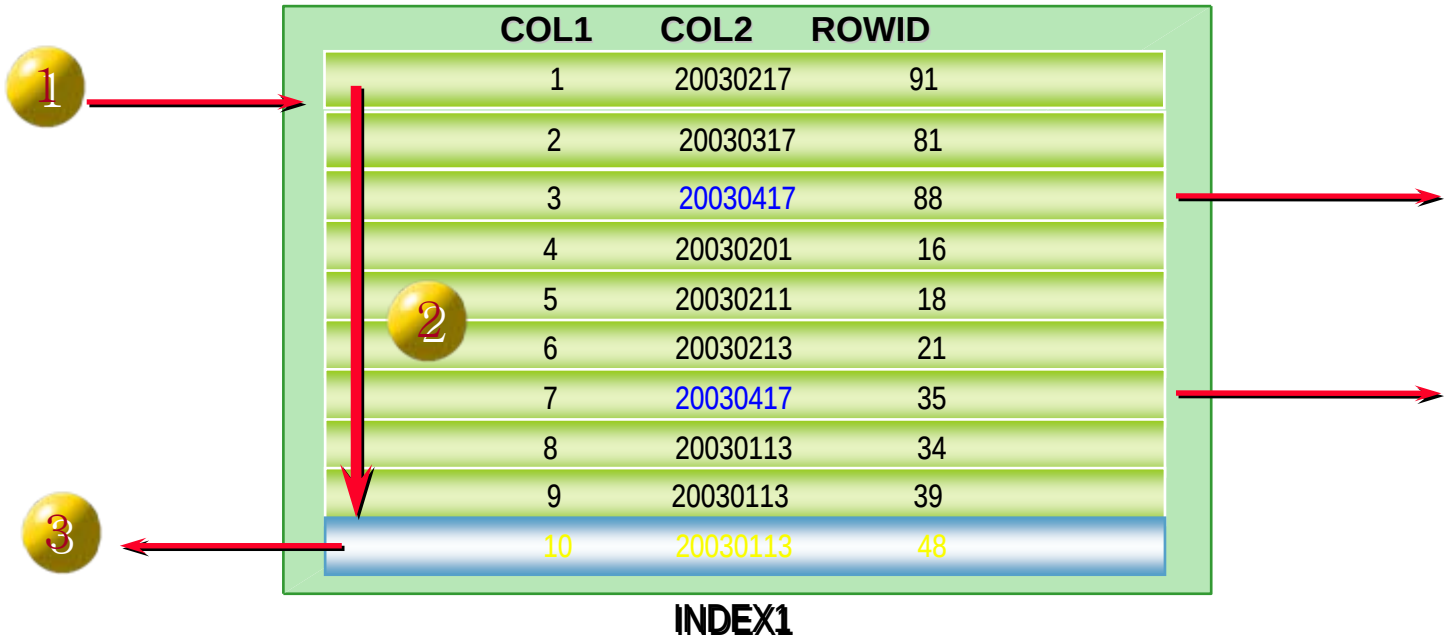
하루 :백만건

```
INSERT STATEMENT, GOAL = FIRST_ROWS
 NESTED LOOPS
  VIEW
   SORT GROUP BY
     TABLE ACCESS BY INDEX ROWID          TKUSERLOG
      INDEX RANGE SCAN          TKUSERLOG_CMN_IN_OUT_IDX
  INDEX UNIQUE SCAN  PK_TKCLIENTVER
```

**TKUSERLOG_CMN_IN_OUT_IDX =**
**Cust_mng_num +**
**logindate+ logoutdate**

## 결합인덱스 ACCESS 원리

```
SELECT * FROM TAB1
WHERE COL1 > 0
    AND COL2 LIKE TO_CHAR(SYSDATE-1,'YYYYMMDD')||'%'
```




```
TABLE ACCESS BY ROWID TAB1
  INDEX RANGE SCAN INDEX1
```

## 개선 가이드

- **TKUSERLOG** 는 하루 평균 **90**만 ~ **100**만건 이상 데이터 발생
- 하루동안 **LOGIN**한 데이터에 대해 분석하는 배치 **JOB**
- 전체 데이터가 **'0'**보다 크므로 **CUST_MNG_NUM > 0** 조건은 데이터 **ACCESS** 범위를 줄이지 못하고, **LOGINDATE,LOGOUTDATE** 역시 **CHECK**조건으로 밖에 쓰일 수 없음.
- 그러므로 전체 데이터가 **5**천만건 이라면 **5**천 만건의 **INDEX**를 **ACCESS**하면서 **LOGINDATE**가 **SYSDATE –1** 인 약 **100** 만 건에 대해 **TABLE RANDOM ACCES** 발생
- 개발자가 **TKUSERLOG_CMN_IN_OUT_IDX** 에 대한 **HINT**를 쓴 이유는 인덱스만 타면 빠르다라는 인덱스에 대한 잘못된 인식 때문

## 개선 방향

- **LOGINDATE**를 선두로 하는 인덱스를 생성할 지라도 하루 데이터가 약 백만건 정도이므로 **RANDOM ACCESS** 부담이 너무 큼
- **LOGINDATE** 를 **PARTITION KEY**로 하여 **PARTITION** 테이블 생성
- **TABLE FULL SCAN**은 **MULTI BLOCK I/O**가능 하므로 특정 일자의 파티션 만을 **FULL SCAN** 유도

## 개선 가이드

### 개선 SQL

```
INSERT INTO TKSTATLOGINCNT
SELECT '01', TO_CHAR(SYSDATE - 1, 'YYYYMMDD'), C.MEDIATYPE, C.EQUIPMENTTYPE, CNT
  FROM (SELECT /*+ FULL(A) */
               '01',MEDIATYPE, EQUIPMENTTYPE, COUNT(*) CNT
          FROM TKUSERLOG A
         WHERE LOGINDATE LIKE TO_CHAR(SYSDATE - 1, 'YYYYMMDD') || '%'
           AND LOGOUTDATE   < TO_CHAR(TRUNC(SYSDATE), 'YYYYMMDD')
         GROUP BY  MEDIATYPE, EQUIPMENTTYPE
       ) A, TKCLIENTVER C
 WHERE A.MEDIATYPE     = C.MEDIATYPE
   AND A.EQUIPMENTTYPE = C.EQUIPMENTTYPE
```

```
EXECUTE PLAN
----------------------------------------------------------------------
INSERT STATEMENT, GOAL = FIRST_ROWS
 NESTED LOOPS
  VIEW
   SORT GROUP BY
    PARTITION RANGE ITERATOR                KEY        KEY
     TABLE ACCESS FULL          TKUSERLOG   KEY        KEY
  INDEX UNIQUE SCAN  PK_TKCLIENTVER
```

## 개선전

```
SELECT T.STATUS,T.MEDIATYPE,A.CUST_NAME,
       R.PROFILE_NICK,T.CADDRESS,T.ONCOMMENT,
       T.FROOMTYPE,T.FROOMTITLE
  FROM  USERTK.TKAUTH T,
       USERAPP.APP_USER_INFO A,
       USERREL.R_PROFILE R
 WHERE T.CUST_MNG_NUM = A.CUST_MNG_NUM
   AND A.CUST_MNG_NUM = R.CUST_MNG_NUM
   AND T.CUST_MNG_NUM = :p_cust_mng_num
```

```
VIEW :
CREATE OR REPLACE VIEW
     USERAPP.APP_USER_INFO AS
SELECT *
  FROM TB_SVC_INFO SVCI,
       TB_CUST_INFO CUST
WHERE SVCI.CUST_MNG_NO =
       CUST.CUST_MNG_NO
```

```
SELECT STATEMENT, GOAL = FIRST_ROWS
 NESTED LOOPS
  NESTED LOOPS
   NESTED LOOPS
    TABLE ACCESS BY INDEX ROWID R_PROFILE
     INDEX UNIQUE SCAN          PK_PROFILE
    TABLE ACCESS BY INDEX ROWID TB_SVC_INFO
     INDEX UNIQUE SCAN          PK_TB_SVC_INFO
   TABLE ACCESS BY INDEX ROWID  TKAUTH
    INDEX UNIQUE SCAN           TKAUTH_PK
  TABLE ACCESS BY INDEX ROWID   TB_CUST_INFO
   INDEX UNIQUE SCAN PK_TB_CUST_INFO
```

## 개선 가이드

- 고객명(CUST_NAME )을 USEREL.R_PROFILE과 USERAPP.TB_CUST_INFO 동시에 관리
- 데이터 중복 및 개발자에게 혼돈
- CUST_NAME 은 USEREL.R_PROFILE 에 존재하므로 USERAPP.APP_USER_INFO 은 불필요한 조인
- 사용자가 로인시에 반드시 수행되는 SQL로서 불필요한 I/O만 발생

### 개선 SQL

```
SELECT T.STATUS,T.MEDIATYPE,R.CUST_NM,
       R.PROFILE_NICK,T.CADDRESS,T.ONCOMMENT,
       T.FROOMTYPE,T.FROOMTITLE
  FROM USERTK.TKAUTH T,
       USERREL.R_PROFILE R
 WHERE T.CUST_MNG_NUM = R.CUST_MNG_NUM
   AND T.CUST_MNG_NUM = :p_cust_mng_num
```

# Q & A